Python Crash Course

Gerth Stølting Brodal
Department of Computer Science
Aarhus University

<table>
<thead>
<tr>
<th>Program</th>
<th>Date</th>
<th>Time</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monday</td>
<td>November 9, 13:00-16:00</td>
<td>I: Python language introduction</td>
<td></td>
</tr>
<tr>
<td>Wednesday</td>
<td>November 11, 14:00-16:00</td>
<td>II: Some hands on, adjusted to how it goes</td>
<td></td>
</tr>
<tr>
<td>Monday</td>
<td>November 23, 13:00-16:00</td>
<td>III: More Python, Numpy, Matplotlib, Pandas, Numba, pypy, mypy, ...</td>
<td></td>
</tr>
</tbody>
</table>
About me

- Gerth Stølting Brodal, professor (PhD Aarhus University 1997, [cv](#))
- Research *is not* AI, ML, Python, Programming languages, ...
- Research *is* Algorithms and Data Structures
  - Basic research on how to make algorithms and data structures efficient
  - Data structures
  - External memory algorithms
  - Computational geometry
- Teaching
  - Algorithms and Data Structures (1st year Computer Science BSc, since 2002)
  - *Introduction to Programming with Scientific Applications (Python)*
    (1st year Data Science BSc, 2nd year Mathematics BSc, since 2018)
Google Colaboratory (Colab)

- [https://colab.research.google.com/](https://colab.research.google.com/)

- Online version of Jupyter documents hosted by Google in the cloud

- Documents stored in your Google Drive

- Allows Python to be executed in the cloud using a web browser
<table>
<thead>
<tr>
<th>1. Introduction to Python</th>
<th>10. Functions as objects</th>
<th>19. Linear programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. Python basics / if</td>
<td>11. Object oriented programming</td>
<td>20. Generators, iterators, with</td>
</tr>
<tr>
<td>9. Recursion and Iteration</td>
<td>18. Multi-dimensional data</td>
<td>27. Final lecture</td>
</tr>
</tbody>
</table>

www.cs.au.dk/~gerth/ipsa20/
Why Python ?
The TIOBE Programming Community index is an indicator of the popularity of programming languages. The index is updated once a month. The ratings are based on the number of skilled engineers world-wide, courses and third party vendors. Popular search engines such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu are used to calculate the ratings. It is important to note that the TIOBE index is not about the best programming language or the language in which most lines of code have been written.

<table>
<thead>
<tr>
<th>Nov 2020</th>
<th>Nov 2019</th>
<th>Change</th>
<th>Programming Language</th>
<th>Ratings</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>↑</td>
<td>C</td>
<td>16.21%</td>
<td>+0.17%</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>↑</td>
<td>Python</td>
<td>12.12%</td>
<td>+2.27%</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>↓</td>
<td>Java</td>
<td>11.68%</td>
<td>-4.57%</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
<td>C++</td>
<td>7.60%</td>
<td>+1.99%</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
<td>C#</td>
<td>4.67%</td>
<td>+0.36%</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td></td>
<td>Visual Basic</td>
<td>4.01%</td>
<td>-0.22%</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td></td>
<td>JavaScript</td>
<td>2.03%</td>
<td>+0.10%</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td></td>
<td>PHP</td>
<td>1.79%</td>
<td>+0.07%</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>↑</td>
<td>R</td>
<td>1.64%</td>
<td>+0.66%</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>↓</td>
<td>SQL</td>
<td>1.54%</td>
<td>-0.15%</td>
</tr>
</tbody>
</table>
Popularity of programming languages

TIODE Programming Community Index

Source: www.tiobe.com

Most Popular Programming Languages 1965 – 2019 (YouTube)
Two Python programs
A Python program

- 7 is an *integer literal* – in Python denoted an “`int`”
- `x` is the name of a *variable* that can hold some value
- `=` is assigning a value to a variable
- `*` denotes multiplication
- `print` is the name of a built-in *function*, here we call `print` to print the result of 7*7
- A program consists of a sequence of *statements*, executed sequentially
Question – What is the result of this program?

Python shell

```python
> x = 3
> y = 5
> x = 2
> print(x * y)
```

x assigned new value

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>old x 3</td>
</tr>
<tr>
<td>y 5</td>
</tr>
<tr>
<td>new x 2</td>
</tr>
</tbody>
</table>

å) 10
b) 15
c) 25
d) [15, 10]
e) Error
f) Don’t know
Another Python program using lists

- [13, 27, 7, 42] is a list containing four integers
- a[2] refers to the entry in the list with index 2 (the first element has index 0, i.e. a[2] is the 3rd element of the list)
- Note that print also can print a list

Python shell
```
> a = [13, 27, 7, 42]
> print(a)
[13, 27, 7, 42]
> print(a[2])
7
```

Memory
```
a
  a[0] 13
  a[1] 27
  a[2] 7
  a[3] 42
```
Question – What is the result of this program?

Python shell

```python
> a = [3, 5, 7]
> print(a[1] + a[2])
```

Memory

```
3  a[0]
5  a[1]
7  a[2]
```

a) 8
b) 10

😊 c) 12
d) 15
e) Don’t know
1 + 2 + \cdots + n

**add.py**

```python
import sys

n = int(sys.argv[1])
sum = 0
for i in range(1, n + 1):
    sum += i
print("Sum = %d" % sum)
```

**add.c**

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int n = atoi(argv[1]);
    int sum = 0;
    for (int i=1; i<=n; i++)
        sum += i;
    printf("Sum = %d\n", sum);
}
```

**add.cpp**

```cpp
#include <iostream>
#include <cstdlib>

using namespace std;

int main(int argc, char *argv[]) {
    int n = atoi(argv[1]);
    int sum = 0;
    for (int i=1; i<=n; i++)
        sum += i;
    cout << "Sum = " << sum << endl;
}
```

**add.java**

```java
class Add{
    public static void main(String args[]){
        int n = Integer.parseInt(args[0]);
        int sum = 0;
        for (int i=1; i<=n; i++)
            sum += i;
        System.out.println("Sum = " + sum);
    }
}
```
Timing results

<table>
<thead>
<tr>
<th>n</th>
<th>C (gcc 9.2)</th>
<th>C++, int (g++ 9.2)</th>
<th>C++, long (g++ 9.2)</th>
<th>Java (12.0)</th>
<th>Python (3.8.1)</th>
<th>PyPy (7.3.0)</th>
<th>Numba, int64</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^7$</td>
<td>0.001 sec*</td>
<td>0.001 sec*</td>
<td>0.003 sec</td>
<td>0.006 sec*</td>
<td>1.5 sec</td>
<td>0.27 sec</td>
<td>0.002 sec</td>
</tr>
<tr>
<td>$10^9$</td>
<td>0.10 sec**</td>
<td>0.10 sec**</td>
<td>0.30 sec</td>
<td>0.40 sec**</td>
<td>145 sec</td>
<td>27 sec</td>
<td>0.2 sec</td>
</tr>
</tbody>
</table>

Wrong output (overflow)

* -2004260032 instead of 50000005000000
** -243309312 instead of 500000000500000000

- since C, C++, and Java only uses 32 bits to represent integers (and 64 bits for ”long” integers)

<table>
<thead>
<tr>
<th>Bit position</th>
<th>bin(10**9)</th>
<th>bin(500000005000000)</th>
<th>bin(-2004260032 + 2**32)</th>
<th>bin(500000000500000000)</th>
<th>bin(-243309312 + 2**32)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11101110011011011001000000000</td>
<td>10110101111001100100010001001001001111011010100101001011000000000</td>
<td>10001000100010010010110101101010000000000</td>
<td>1101111000001011011010110110011111000101111111110110101000000000</td>
<td>111100001011111110110101100000000000</td>
</tr>
</tbody>
</table>

Try Google: civilization.gandhi overflow
## Timing results

<table>
<thead>
<tr>
<th>n</th>
<th>C (gcc 9.2)</th>
<th>C++, int (g++ 9.2)</th>
<th>C++, long (g++ 9.2)</th>
<th>Java (12.0)</th>
<th>Python (3.8.1)</th>
<th>PyPy (7.3.0)</th>
<th>Numba, int64</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^7$</td>
<td>0.001 sec*</td>
<td>0.001 sec*</td>
<td>0.003 sec</td>
<td>0.006 sec*</td>
<td>1.5 sec</td>
<td>0.27 sec</td>
<td>0.002 sec</td>
</tr>
<tr>
<td>$10^9$</td>
<td>0.10 sec**</td>
<td>0.10 sec**</td>
<td>0.30 sec</td>
<td>0.40 sec**</td>
<td>145 sec</td>
<td>27 sec</td>
<td>0.2 sec</td>
</tr>
</tbody>
</table>

- **Relative speed**

  \[ C \approx C++ > Java >> Python \]

- C, C++, Java need to care about integer overflows – select integer representation carefully with sufficient number of bits (8, 16, 32, 64, 128)

- Python natively works with arbitrary long integers (as memory on your machine allows). Also possible in Java using the class `java.math.BigInteger`

- Python programs can (sometimes) run faster using PyPy

- Number crunching in **Python** should be delegated to **specialized modules** (e.g. **Numpy**, **CPLEX**, **Numba**) – often written in C or C++
Why Python?

- Short concise code
- Index out of range exceptions
- Elegant for-each loop
- Garbage collection is done automatically
- Exact integer arithmetic (no overflows)
- Can delegate number crunching to C, C++, ...
History of Python development

- Python created by Guido van Rossum in 1989, first release 0.9.0 1991
- **Python 2 → Python 3** (clean up of Python 2 language)
  - Python 2 – version 2.0 released 2000, final version 2.7 released mid-2010
  - Python 3 – released 2008, current release 3.8.1
- Python 3 is *not* backward compatible, libraries incompatible

<table>
<thead>
<tr>
<th>Python 2</th>
<th>Python 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>print 42</td>
<td>print(42)</td>
</tr>
<tr>
<td>int = C long (32 bits)</td>
<td>int = arbitrary number of digits (= named “long” in Python 2)</td>
</tr>
<tr>
<td>7/3 → 2 returns “int”</td>
<td>7/3 → 2.333... returns “float”</td>
</tr>
<tr>
<td>range() returns list (memory intensive)</td>
<td>range() returns iterator (memory efficient; xrange in Python 2)</td>
</tr>
</tbody>
</table>

100th episode of Talk Python To Me: *Python past, present, and future with Guido van Rossum*
Python.org
Running the Python Interpreter

- Open Command Prompt (Windows-key + cmd)
- Type “python” + return
- Start executing Python statements

- To exit shell: Ctrl-Z + return or exit() + return
Some other usefull packages

- Try installing some more Python packages:

  - `pip install numpy`  
    linear algebra support (N-dimensional arrays)
  - `pip install scipy`  
    numerical integration and optimization
  - `pip install matplotlib`  
    2D plotting library
  - `pip install pylint`  
    Python source code analyzer enforcing a coding standard
Creating a Python program the very basic way

- Open Notepad
  - write a simple Python program
  - save it
- Open a command prompt
  - go to folder (using cd)
  - run the program using
    \texttt{python <program name>.py}
... or open IDLE and run program with F5

```python
x = 3
y = 4
print(x * y)
```

```
12
```
The Python Ecosystem

- **Interpreters/compiler**
  - CPython – reference C implementation from python.org
  - PyPy – written in RPython (a subset of Python) – faster than Cpython
  - Jython – written in Java and compiles to Java bytecode, runs on the JVM
  - IronPython – written in C#, compiles to Microsoft’s Common Language Runtime (CLR) bytecode
  - Cython – project translating Python-ish code to C

- **Shells (IPython, IDLE)**

- **Libraries/modules/packages**
  - pypi.python.org/pypi (PyPI - the Python Package Index, +200,000 packages)

- **IDEs (Integrated development environment)**
  - IDLE comes with Python (docs.python.org/3/library/idle.html)
  - Anaconda w. Spyder, IPython (www.anaconda.com/download)
  - Canopy (enthought.com/product/canopy)
  - Python tools for Visual Studio (github.com/Microsoft/PTVS)
  - PyCharm (www.jetbrains.com/pycharm/)
  - Emacs (Python mode and ElPy mode)
  - Notepad++

- **Python Style guide (PEP8)**
  - pylint, pep8, flake8
Python basics

- Comments
- ";;"
- Variable names
- int, float, str
- type conversion
- assignment (=)
- print(), help(), type()
Python comments

A ‘#’ indicates the beginning of a comment. From ‘#’ until of end of line is ignored by Python.

```python
x = 42  # and here goes the comment
```

Comments useful to describe what a piece of code is supposed to do, what kind of input is expected, what is the output, side effects...
The “;” in Python

- Normally statements follow in consecutive lines with identical indentation
  
  ```python
  x = 1
  y = 1
  ```

- but Python also allows multiple statements on one line, separated by “;”
  
  ```python
  x = 1; y = 1
  ```

- General Python PEP 8 guideline: avoid using “;”

- Other languages like C, C++ and Java require “;” to end/separate statements

neither `pylint` or `flake8` like “;”
Variable names

- Variable name = sequence of **letters** ‘a’–’z’, ‘A’–’Z’, **digits** ‘0’–’9’, and **underscore** ‘_’
  - v, volume, height_of_box, WidthOfBox, x0, _v12_34B, _
    - (snake_case)
    - (CamelCase)
  - a name cannot start with a digit
  - names are case sensitive (AB, Ab, aB and ab are different variables)

- Variable names are **references to objects in memory**

- **Use meaningful variables names**

- **Python 3 reserved keywords:** and, as, assert, break, class, continue, def, del, elif, else, except, False, finally, for, from, global, if, import, in, is, lambda, nonlocal, None, not, or, pass, raise, return, True, try, while, with, yield
Question – Not a valid Python variable name?

a) print

![Sad face] b) for  
   Python reserved keyword

c) _100

d) x

e) _

f) python_for_ever

g) Don’t know

Python shell

```python
> print = 7
> print(42)
traceback (most recent call last):
  File "<stdin>" line 1 in <module>
TypeError: \'int\' object is not callable
```

print is a valid variable name, with default value a built-in function to print output to a shell – assigning a new value to print is very likely a bad idea (like many others sum, int, str, ...)
Integer literals

- ..., −4, −3, −2, −1, 0, 1, 2, 3, 4, ...

- Python integers can have an arbitrary number of digits (only limited by machine memory)

- Can be preceded by a plus (+) or minus (−)

- For readability underscores (_) can be added between digits, 2_147_483_647

(for more, see PEP 515 - Underscores in Numeric Literals)
Question – What statement will not fail?

a) \( x = _42 \)
b) \( \_10 = -1_1 \)
\( \text{😊} \)
c) \( x = 1__0 \)
d) \( x = +1_0_0 \)
e) Don’t know
Float literals

- Decimal numbers are represented using float – contain “.” or “e”
- Examples
  - 3.1415
  - -.00134
  - 124e3 = 124 \times 10^3
  - -2.345e2 = -234.5
  - 12.3e-4 = 0.00123

- Floats are often only approximations, e.g. 0.1 is not 1/10

- Extreme values (CPython 3.8.1)
  - max = 1.7976931348623157e+308
  - min = 2.2250738585072014e-308

- NB: Use module fractions for exact fractions/rational numbers

Python shell

```python
> 0.1 + 0.2 + 0.3
| 0.6000000000000001
> (0.1 + 0.2) + 0.3
| 0.6000000000000001
> 0.1 + (0.2 + 0.3)
| 0.6
> type(0.1)
| <class 'float'>
> 1e200 * 1e300
| inf
> 0.1+(0.2+0.3) == (0.1+0.2)+0.3
| False
> x = 0.1 + 0.2
> y = 0.3
> x == y
| False
> print(f'{x:.30f}')  # 30 decimals
| 0.300000000000000044408920985006
> print(f'{y:.30f}')  # 30 decimals
| 0.299999999999999988897769753748
> import sys
> sys.float_info.min
| 2.2250738585072014e-308
> sys.float_info.max
| 1.7976931348623157e+308
```
Question – What addition order is ”best”?

a) \(1e10 + 1e-10 + -5e-12 + -1e10\)

b) \(1e10 + -1e10 + 1e-10 + -5e-12\)

c) \(1e-10 + 1e10 + -1e10 + -5e-12\)

d) \(-5e-12 + -1e10 + 1e10 + 1e-10\)

e) Any order is equally good

f) Don’t know

\[
\begin{align*}
1e10 & = 10000000000 \\
-le10 & = -10000000000 \\
1e-10 & = 0.0000000001 \\
-5e-12 & = -0.00000000005
\end{align*}
\]

Python shell

```
> 1e10 + 1e-10 + -5e-12 + -1e10
| 0.0
> 1e10 + -1e10 + 1e-10 + -5e-12
| 9.5000000000000001e-11
> 1e-10 + 1e10 + -1e10 + -5e-12
| -5e-12
> -5e-12 + -1e10 + 1e10 + 1e-10
| 1e-10
```

a) - d) give four different outputs
Approximating
\[
\pi = 3.14159265359...
\]

\[
\frac{\pi^2}{6} = \sum_{k=1}^{+\infty} \frac{1}{k^2} = 1.6449340668...
\]

Riemann zeta function \( \zeta(2) \)

This is not a course in numeric computations – but now you are warned....

```
pi_approximation_riemann.py

```
Python float ≡ IEEE-754 double precision*

- A binary number is a number in base 2 with digits/bits from \{0,1\}
  \[
  10110_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 16 + 4 + 2 = 22_{10}
  \]

- IEEE-754 64-bit double

<table>
<thead>
<tr>
<th>sign $s$</th>
<th>exponent $e$</th>
<th>coefficient $c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>11 bits</td>
<td>52 bits</td>
</tr>
</tbody>
</table>

Float value Case

<table>
<thead>
<tr>
<th>$(-1)^s \cdot (1 + c \cdot 2^{-52}) \cdot 2^{e-1023}$</th>
<th>$0 &lt; e &lt; 2047$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(-1)^s \cdot c \cdot 2^{-1074}$</td>
<td>$e = 0, c \neq 0$</td>
</tr>
<tr>
<td>+0 and -0</td>
<td>$e = 0, c = 0$</td>
</tr>
<tr>
<td>+∞ and -∞</td>
<td>$e = 2047, c = 0$</td>
</tr>
<tr>
<td>NaN (“not a number”) s = 0, e = 2047, c ≠ 0</td>
<td></td>
</tr>
</tbody>
</table>

(*most often, but there is no guarantee given in the Python language specification that floats are represented using IEEE-754)
String literals (type `str`)

- **Sequence of characters** enclosed by single (') or double ("") quotes
  
  "a 'quoted' word"   "Hello World"   'abc'
  'a "quoted" word'   '_"_"_'

- **Escape characters**
  
  \n  newline
  \t  tab
  \  backslash
  '  single quote
  "  double quote

- A backslash (\) a the end of line, will continue line/string on next line

- **Use triple single or double quotes** ( ' ' ' or " "") for enclosing strings spanning more lines
  
  (in particular for Python Docstrings, see [PEP 257])
Question – What does the following print?

```python
print("\\\\\\n\n\n'")
```

a) \\\n\n\n'
b) \\
nn'
c) \\
n
'
d) "nn'
e) \
'
f) Don’t know
Long string literals

- Long string literals often need to be split over multiple lines
- In Python two (or more) string literals following each other will be treated as a single string literal (they can use different quotes)
- Putting parenthesis around multiple literals allows line breaks
- Advantages:
  - avoids the backslash at the end of line
  - can use indentation to increase readability
  - allows comments between literals

```
long-string-literals.py

s1 = 'abc' "def" # two string literals
print(s1)
s2 = '"' '"' '"' # avoid escaping quotes
print(s2)
s3 = 'this is a really, really, really, \really, really, long string'
print(s3)
s4 = ('this is a really, really, '
     'really, really, really, '
     'long string')
print(s4)
very_very_long_variable_name = (
    'this is a really, really, '
    'really, really, really, '
    'long string'
)
print(very_very_long_variable_name)
```

Python shell
```
| abcdef
| ""
| this is a really, really, really, really, long string
| this is a really, really, really, really, long string
| this is a really, really, really, realy, really, long string
```
Raw string literals

- By prefixing a string literal with an `r`, the string literal will be considered a raw string and backslashes become literal characters.
- Useful in cases where you actually need backslashes in your strings, e.g. when working with Python’s regular expression module `re`.

```python
print('\let\epsilon\varepsilon')  # \v = vertical tab
print('\\let\\epsilon\\varepsilon')  # many backslashes
print(r'\let\epsilon\varepsilon')  # more readable
```
print(...)  

- **print** can print zero, one, or more values  
- **default behavior**  
  - print a space between values  
  - print a line break after printing all values  
- **default behavior can be changed by keyword arguments** “sep” and “end”
print(...) and help(...)
Assignments

- **variable = expression**
  
  \[
  x = 42
  \]

- Multiple assignments – right hand side evaluated before assignment
  
  \[
  x, y, z = 2, 5, 7
  \]

- Useful for swapping
  
  \[
  x, y = y, x
  \]

- Assigning multiple variables same value in left-to-right
  
  \[
  x = y = z = 7
  \]

---

**Warning**

\[
\begin{align*}
  i &= 1 \\
  i &= v[i] = 3 \quad # v[3] \text{ is assigned value } 3
\end{align*}
\]

In languages like C and C++ instead

\[
  v[1] \text{ is assigned } 3
\]
Python is dynamically typed, type(...)

- The current type of a value can be inspected using the `type()` function (that returns a type object)
- In Python the values contained in a variable over time can be of different type
- In languages like C, C++ and Java variables are declared with a given type, e.g.
  ```python
  int x = 42;
  ```
  and the different values stored in this variable must remain of this type

```
Python shell

> x = 1
> type(x)
| <class 'int'>
> x = 'Hello'
> type(x)
| <class 'str'>
> type(42)
| <class 'int'>
> type(type(42))
| <class 'type'>
```

x new type
Type conversion

- Convert a value to another type: \( \text{new-type}(\text{value}) \)

- Sometimes done automatically:

\[
1.0 + 7 = 1.0 + \text{float}(7) = 8.0
\]
Questions – \texttt{str(float(int(float("7.5"))))} ?

a) 7
b) 7.0
c) 7.5
d) "7"

😀 e) "7.0"

f) "7.5"
g) Don’t know
Control structures

- `input()`
- `if-elif-else`
- `while-break-continue`
input

- The built-in function `input(message)` prints `message`, and waits for the user to provide a line of input and press return. The line of input is returned as a `str`.

- If you e.g. expect input to be an `int`, then remember to convert the input using `int()`.

```python
name-age.py
name = input('Name: ')
age = int(input('Age: '))
print(name, 'is', age, 'years old')
```

Python shell

```sh
> Name: Donald Duck
> Age: 84
| Donald Duck is 84 years old
```
Branching – do either this or that?

Code before

make decision?

Boolean expression

True

do this

False

do that

Code after
Basic if-else

```python
if boolean expression:
  code
else:
  code
```

Identical indentation for a sequence of lines = the same spaces/tabs should precede code
pass

- **pass** is a Python statement doing nothing. Can be used where a statement is required but you want to skip (e.g. code will be written later).

- Example (bad example, since `else` could just be omitted):

```python
if x % 2 == 0:
    print('even')
else:
    pass
```
if-elif-else

if condition:
    code
elif condition: # zero or more “elfi“ = “else if”
    code
else: # optional
    code

if (condition) {
    code
} else if (condition) {
    code
} else {
    code
}

Java, C, C++ syntax

if.py

```python
if x == 0:
    print('zero')
```

if-else.py

```python
if x % 2 == 0:
    print('even')
else:
    print('odd')
```

elif.py

```python
if x < 0:
    print('negative')
elif x == 0:
    print('zero')
elif x == 1:
    print('one')
else:
    print('>= 2')
```

Other languages using indentation for blocking:
ABC (1976), occam (1983), Miranda (1985)
Questions – What value is printed?

```python
x = 1
if x == 2:
    x = x + 1
else:
    x = x + 1
    x = x + 1
x = x + 1
print(x)
```
a) 1
b) 2
c) 3

✔️ d) 4
e) 5
f) Don’t know
Nested if-statements

```python
if x < 0:
    print('negative')
elif x % 2 == 0:
    if x == 0:
        print('zero')
    elif x == 2:
        print('even prime number')
    else:
        print('even composite number')
else:
    if x == 1:
        print('one')
    else:
        print('some odd number')
```
if-else expressions

- A very common computation is

```
if test:
    x = true-expression
else:
    x = false-expression
```

- In Python there is a shorthand for this:

```
x = true-expression if test else false-expression
```

(see What’s New in Python 2.5 - PEP 308: Conditional Expressions)

- In C, C++ and Java the equivalent notation is (note the different order)

```
x = test ? true-expression : false-expression
```
Repeat until done

Code before

repeat ?
boolean expression

True

do it once more

False

Code after

b
while-statement

```python
while condition:
    code
    ...
break  # jump to code after while loop
...
continue  # jump to condition at the...
    # beginning of while loop
```

The function `randint(a, b)` from module `random` returns a random integer from `{a, a + 1,..., b – 1, b}`

**count.py**
```
x = 1
while x <= 5:
    print(x, end=' ')
    x = x + 1
print('and', x)
```  

**Python shell**
```
1 2 3 4 5 and 6
```
Computing $\lfloor \sqrt{x} \rfloor$ using binary search

```python
int-sqrt.py

x = 20
low = 0
high = x + 1

while True:
    # low <= sqrt(x) < high
    if low + 1 == high:
        break
    mid = (high + low) // 2
    if mid * mid <= x:
        low = mid
        continue
    high = mid

print(low)  # low = floor(sqrt(x))
```

An exercise asks to simplify the code.
Division using the Newton-Raphson method

- **Goal:** Compute $1 / n$ only using $+, -, \text{ and } *$
- $x = 1 / n \iff f(x) = n - 1 / x = 0$
- Problem reduces to finding root of $f$
- **Newton-Raphson:**
  
  $x := x - f(x)/f'(x) = x - (n-1/x)/(1/x^2) = (2-n \cdot x) \cdot x$
  
  since $f'(x) = 1 / x^2$ for $f(x) = n - 1 / x$

```python
division.py
n = 0.75  # n in [0.5, 1.0]
x = 1.0
last = 0.0
while last < x:
    print(x)
    last = x
    x = (2 - n * x) * x
print('Apx of 1.0 /', n, '=', x)
print('Python 1.0 /', n, '=' , 1.0 / n)
```

```
| 1.0 |
| 1.25 |
| 1.328125 |
| 1.33331298828125 |
| 1.3333333330228925 |
| 1.3333333333333333 |

Apx of 1.0 / 0.75 = 1.3333333333333333
Python 1.0 / 0.75 = 1.3333333333333333
```

en.wikipedia.org/wiki/Newton’s_method
Operations

- None, bool
- basic operations
- strings
- += and friends
NoneType

- The type None has only one value: None

- Used when context requires a value, but none is really available

- **Example**: All functions must return a value. The function `print` has the *side-effect* of printing something to the standard output, but returns None

```
Python shell
> x = print(42)
| 42
> print(x)
| None
```

- **Example**: Initialize a variable with no value, e.g. list entries `mylist = [None, None, None, None]`
Type bool

- The type `bool` only has two values: **True** and **False**

- Logic truth tables:

<table>
<thead>
<tr>
<th>x or y</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x and y</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>not x</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>
Scalar vs Non-scalar Types

- **Scalar types** (atomic/indivisible): `int`, `float`, `bool`, `None`
- **Non-scalar**: Examples `strings` and `lists`

```
"string"[3] = "i"
[2, 5, 6, 7][2] = 6
```
Questions – What is \([7, 3, 5][[1, 2, 3][1]]\) ?

a) 1
b) 2
c) 3

\[\boxed{d) 5}\]
e) 7

f) Don’t know
Operations on int and float

Result is float if and only if at least one argument is float, except ** with negative exponent always gives a float

- +, -, * addition multiplication, e.g. 3.0 * 2 = 6.0
- ** and pow(x, y) power, e.g. 2 ** 3 = pow(2, 3) = 8, 2 ** -2 = 0.25
- // integer division = [x / y]
  e.g. 15.0 // 4 = 3.0. Note: -8 // 3 = -3
- / division returns float, 6 / 3 = 2.0
- abs(x) absolute value
- % integer division remainder (modulo)
  11 % 3 = 2
  4.7 % 0.6 = 0.5000000000000003

Python shell

```python
>>> 0.4 // 0.1 |
| 4.0 |
>>> 0.4 / 0.1 |
| 4.0 |
>>> 0.3 // 0.1 |
| 2.0 |
>>> 0.3 / 0.1 |
| 2.9999999999999996 |
>>> 10 ** 1000 / 2 |
| OverflowError: integer division result too large for a float |
```
Running time for $3**x \div 3**x$

Working with larger integers takes slightly more than linear time in the number of digits.
import matplotlib.pyplot as plt

bits, compute_time = [], []

for i in range(42):
    x = 3**i // 2**i
    start = time()
    result = 3**x // 3**x  # the computation we time
    end = time()
    t = end - start
    print("i =", i, "x =", x, "Result =", result, "time(sec) =", t)
    bits.append(x)
    compute_time.append(t)

plt.title('Computing 3**x // 3**x')
plt.xlabel('x')
plt.ylabel('computation time (seconds)')
plt.plot(bits, compute_time, "g:"
plt.plot(bits, compute_time, "ro")
plt.show()
module math

Many standard mathematical functions are available in the Python module “math”, e.g. sqrt, sin, cos, tan, asin, acos, atan, log(natural), log10, exp, ceil, floor, ...

- To use all the functions from the math module use `import math`
  Functions are now available as e.g. `math.sqrt(10)` and `math.ceil(7.2)`
- To import selected functions you instead write `from math import sqrt, ceil`
- The library also contains some constants, e.g. `math.pi = 3.141592...` and `math.e = 2.718281...`
- Note: `x ** 0.5` significantly faster than `sqrt(x)`

Python shell

```python
> (0.1 + 0.2) * 10
| 3.0000000000000004
> math.ceil((0.1 + 0.2) * 10)
| 4
```

docs.python.org/3/library/math.html
Rounding up integer fractions

- Python: \( \lceil \frac{x}{y} \rceil = -(-\frac{x}{y}) \)

<table>
<thead>
<tr>
<th>Python</th>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-(13/3) = 5</td>
<td>-(13/3) = 4</td>
<td>-(13/3) = 4</td>
</tr>
</tbody>
</table>

⚠️ The intermediate result \( \frac{x}{y} \) in `math.ceil(x/y)` is a float with limited precision

- Alternative computation:
  \[
  \lceil \frac{x}{y} \rceil = \frac{x + (y-1)}{y}
  \]
There exists special float values inf, -inf, nan representing “+infinity”, “-infinity” and “not a number”

Can be created using e.g. float('inf') or imported from the math module

Some overflow operations generate an OverflowError, other return inf and allow calculations to continue!

Read the IEEE 754 standard if you want to know more details...

---

Python shell

```python
> 1e250 ** 2
OverflowError: (34, 'Result too large')
> 1e250 * 1e250
inf
> -1e250 * 1e250
-inf
> import math
> math.inf
inf
> type(math.inf)
<class 'float'>
> math.inf / math.inf
nan
> type(math.nan)
<class 'float'>
> math.nan == math.nan
false
> float('inf') - float('inf')
nan
```
Operations on bool

- The operations **and**, **or**, and **not** behave as expected when the arguments are False/True.
- The three operators also accept other types, where the following values are considered `false`:

  False, None, 0, 0.0, "", [], ...

(see The Python Standard Library > 4.1. True Value Testing for more false values)

**Short-circuit evaluation**: The rightmost argument of **and** and **or** is only evaluated if the result cannot be determined from the leftmost argument alone. The result is either the leftmost or rightmost argument (see truth tables), i.e. the result is not necessarily False/True.

True or 7/0 is completely valid since 7/0 will never be evaluated (which otherwise would throw a ZeroDivisionError exception)
Questions – What is "abc" and 42?

a) False
b) True
c) "abc"
d) 42

✿
e) TypeError
f) Don’t know
Comparison operators (e.g. int, float, str)

==  test if two objects are equal, returns bool
    not to be confused with the assignment operator (=)
!=  not equal
>
>=
<
<=
Chained comparisons

- A recurring condition is often
  \[ x < y \text{ and } y < z \]

- If \( y \) is a more complex expression, we would like to avoid computing \( y \) twice, i.e. we often would write
  \[
  \text{tmp} = \text{complex expression} \\
  x < \text{tmp} \text{ and } \text{tmp} < z
  \]

- In Python this can be written as a chained comparisons (which is shorthand for the above)
  \[ x < y < z \]

- Note: Chained comparisons do not exist in C, C++, Java, ...
Questions – What is $1 < 0 < \frac{6}{0}$?

a) True

b) False

c) 0

d) 1

e) 6

f) ZeroDivisionError

g) Don’t know
Binary numbers and operations

- Binary number = integer written in base 2: \(101010_2 = 42_{10}\)

- Python constant prefix 0b: \(0b101010 \rightarrow 42\)
- \(\text{bin}(x)\) converts integer to string: \(\text{bin}(49) \rightarrow "0b110001"\)
- \(\text{int}(x, 2)\) converts binary string value to integer: \(\text{int}("0b110001", 2) \rightarrow 49\)

- Bitwise operations
  - | Bitwise OR
  - & Bitwise AND
  - ~ Bitwise NOT (\(\sim x\) equals to \(-x - 1\))
  - ^ Bitwise XOR

- Example: \(\text{bin}(0b1010 \mid 0b1100) \rightarrow "0b1110"\)

- Hexadecimal = base 16, Python prefix 0x: \(0x30 \rightarrow 48, 0xA0 \rightarrow 160, 0xFF \rightarrow 255\)

- \(<<<\) and \(>>\) integer bit shifting left and right, e.g. \(12 \ggg 2 \rightarrow 3\), and \(1 \ll 4 \rightarrow 16\)
Operations on strings

- `len(str)` returns length of `str`
- `str[index]` returns `index+1`'th symbol in `str`
- `str1 + str2` returns concatenation of two strings
- `int * str` concatenates `str` with itself `int` times

Formatting: `% operator` or `.format()` function

Old Python 2 way since Python 3.0

Or formatted string literals (f-strings) with prefix letter `f` and Python expressions in `{ }`

(see `pyformat.info` for an introduction)

From “What’s New In Python 3.0”, 2009: A new system for built-in string formatting operations replaces the `% string formatting operator. (However, the % operator is still supported; it will be deprecated in Python 3.1 and removed from the language at some later time.) Read PEP 3101 for the full scoop.
... more string functions

- `str[−index]` returns the symbol i positions from the right, the rightmost `str[−1]`
- `str[from:to]` substring starting at index `from` and ending at index `to-1`
- `str[from:−to]` substring starting at `form` and last at index `len(str) − to − 1`
- `str[from:to:step]` only take every `step`’th symbol in `str[from:to]`
  
  - `from` or/and `to` can be omitted and defaults to the beginning/end of string
- `chr(x)` returns a string of length 1 containing the `x`’th Unicode character
- `ord(str)` for a string of length 1, returns the Unicode number of the symbol
- `str.lower()` returns string in lower case
- `str.split()` split string into list of words, e.g.
  
  "we love python".split() = ['we', 'love', 'python']

[docs.python.org/3/library/string.html](docs.python.org/3/library/string.html)
Questions – What is $s[2:42:3]$?

\[ s = 'ab\textcolor{red}{wwdexy___lwtopavghevt_xypxxxyattx_hxwoadnxxx}' \]

a) $'\textcolor{red}{wwdexy___lwtopavghevt_xypxxxyattx_hxwoadn}'$

b) $'\textcolor{red}{we\_love\_python}'$

c) $'\textcolor{red}{we\_love\_java}'$

d) Don’t know
Strings are immutable

- Strings are non-scalar, i.e. for $s = \"abcdef\", s[3] \text{ will return } \"d\"

- Strings are immutable and cannot be changed once created. I.e. the following natural update is not possible (but is e.g. allowed in C)

  $$s[3] = \"x\"$$

- To replace the \"d\" with \"x\" in $s$, instead do the following update

  $$s = s[:3] + \"x\" + s[4:]$$
Operators
Precedence rules & Associativity

Example: * has higher precedence than +

\[ 2 + 3 * 4 \equiv 2 + (3 * 4) \rightarrow 14 \quad \text{and} \quad (2 + 3) * 4 \rightarrow 20 \]

All operators in same group are evaluated left-to-right

\[ 2 + 3 - 4 - 5 \equiv ((2 + 3) - 4) - 5 \rightarrow -4 \]

except for **, that is evaluated right-to-left

\[ 2**2**3 \equiv 2**(2**3) \rightarrow 256 \]

Rule: Use \textit{parenthesis} whenever in doubt of precedence!

\[ \text{docs.python.org/3/reference/expressions.html} \]
Long expressions

- Long expressions can be broken over several lines by putting parenthesis around it.
- The PEP8 guidelines recommend to limit all lines to a maximum of 79 characters.

Python shell

```
> (1 + 2 + 3)
6
```

https://www.python.org/dev/peps/pep-0008/#maximum-line-length
+= and friends

- Recurring statement is

\[ x = x + \text{value} \]

- In Python (and many other languages) this can be written as

\[ x += \text{value} \]

- This also applies to other operators like

\[ += \quad -= \quad *= \quad /= \quad //= \quad **= \]
\[ |= \quad &= \quad ^= \quad <<= \quad >>= \]

Python shell:

```
> x = 5
> x *= 3
> x
| 15
> a = 'abc'
> a *= 3
> a
| 'abcabcabc'
```
:=  assignment expressions (the “Walrus Operator”)

- **Syntax**
  
  name := expression

- Evaluates to the value of expression, with the side effect of assigning result to name

- Useful for naming intermediate results/repeating subexpressions for later reusage

- See [PEP 572](https://www.python.org/dev/peps/pep-0572/) for further details and restrictions of usage

- In some languages, e.g. Java, C and C++, “=” also plays the role of “:=“, implying “if (x=y)” and “if (x==y)” mean quite different things (common typo)
Lists

- List syntax
- List operations
- copy.deepcopy
- range
- while-else
- for
- for-break-continue-else
List operations

- List syntax \([value_1, \ value_2, \ldots, \ value_k]\)
- List indexing \(L[index], \ L[-index]\)
- List slices \(L[from:to], \ L[from:to:step]\) or \(L[slice(from, to, step)]\)
- Creating a copy of a list \(L[:]\) or \(L.copy()\)
- List concatenation (creates new list) \(X + Y\)
- List repetition (repeated concatenation with itself) \(42 * L\)
- Length of list \(\text{len}(L)\)
- Check if element is in list \(e \ in \ L\)
- Index of first occurrence of element in list \(L.index(e)\)
- Number of occurrences of element in list \(L.count(e)\)
- Check if element is not in list \(e \ not \ in \ L\)
- \(\text{sum}(L) \ \text{min}(L) \ \text{max}(L)\)

[docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range]
List modifiers (lists are mutable)

- Extend list with elements (X is modified)  X.extend(Y)
- Append an element to a list (L is modified)  L.append(42)
- Replace sublist by another list (length can differ)  X[i:j] = Y
- Delete elements from list  del L[i:j:k]
- Remove & return element at position  L.pop(i)
- Remove first occurrence of element  L.remove(e)
- Reverse list  L.reverse()
- L *= 42
- L.insert(i, x) same as L[i:i] = [x]
Questions – What is $x$?

$x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$

$x[2:8:3] = [\text{\textquotesingle}a\text{\textquotesingle}, \text{\textquotesingle}b\text{\textquotesingle}]$

a) $[1, 2, \text{\textquotesingle}a\text{\textquotesingle}, \text{\textquotesingle}b\text{\textquotesingle}, 5, 6, 7, 8, 9, 10]$

b) $[1, \text{\textquotesingle}a\text{\textquotesingle}, 3, 4, 5, 6, 7, \text{\textquotesingle}b\text{\textquotesingle}, 9, 10]$

c) $[1, 2, 3, 4, 5, 6, 7, \text{\textquotesingle}a\text{\textquotesingle}, \text{\textquotesingle}b\text{\textquotesingle}]$

$d) [1, 2, \text{\textquotesingle}a\text{\textquotesingle}, 4, 5, \text{\textquotesingle}b\text{\textquotesingle}, 7, 8, 9, 10]$

e) ValueError

f) Don’t know
Questions – What is \( y \)?

\[
y = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
y = y[3:15:3][1:4:2]
\]

a) \([3, 6, 9, 12, 15]\)

b) \([7, 13]\)

c) \([1, 9]\)

d) \([4, 7, 10, 13, 2, 4]\)

e) TypeError

f) Don’t know
Nested lists (multi-dimensional lists)

- Lists can contain lists as elements, that can contain lists as elements, that ...
- Can e.g. be used to store multi-dimensional data (list lengths can be non-uniform)

Note: For dealing with matrices the **NumPy** module is a better choice

```python
list1d = [1, 3, 5, 2]
list2d = [[1, 2, 3, 4],
          [5, 6, 7, 9],
          [0, 8, 2, 3]]
list3d = [[[5, 6], [4, 2], [1, 7], [2, 4]],
          [[1, 2], [6, 3], [2, 5], [7, 5]],
          [[3, 8], [1, 5], [4, 3], [2, 4]]]
print(list1d[2])
print(list2d[1][2])
print(list3d[2][0][1])
```
aliasing

\[ a = [13, 27, 7, 42] \]
\[ b = a \]
\[ a[2] = 12 \]
$y = x \text{ VS } y = x[:]$ 

$a = [13, 27, 7, 42]$

$b = a$

$a[2] = 12$

$a = [13, 27, 7, 42]$

$b = a[:]$

$a[2] = 12$
x [ :: ] vs nested structures

```python
a = [[3,5],[7,11]]
b = a
c = a[:]
a[0][1] = 4
c[1] = b[0]
```
Question – what is \( c \) ?

a) \([ [3, 5], [7, 11] ]\)

b) \([ [3, 5], [3, 5] ]\)

c) \([ [3, 4], [3, 5] ]\)

d) \([ [3, 4], [3, 4] ]\)

e) Don’t know

\[
a = \begin{bmatrix} [3, 5], [7, 11] \end{bmatrix}
\]

\[
b = a
\]

\[
c = a[:]
\]

\[
a[0][1] = 4
\]

\[
c[1] = b[0]
\]
To make a copy of all parts of a composite value use the function `deepcopy` from module `copy`.

```python
from copy import deepcopy
a ==[[3, 5], 7]
b = deepcopy(a)
a[0][0] = 4
print(a)  # [[4, 5], 7]
print(b)  # [[3, 5], 7]
```
Initializing a 2-dimensional list

Python shell

```
> x = [1] * 3
> x
[1, 1, 1]

> y = [[1] * 3] * 4
> y
[[[1, 1, 1], [1, 1, 1], [1, 1, 1], [1, 1, 1]]

> y[0][0] = 0
> y
[[0, 1, 1], [0, 1, 1], [0, 1, 1], [0, 1, 1]]
```

Python shell

```
> y = []
> for _ in range(4): y.append([1] * 3)
> y[0][0] = 0
> y
[[0, 1, 1], [1, 1, 1], [1, 1, 1], [1, 1, 1]]
```
range\( (\text{from}, \text{to}, \text{step}) \)  

- **range** \((\text{from}, \text{to}, \text{else})\) generates the sequence of numbers starting with \text{from}, with increments of \text{step}, and smaller/greater than \text{to} if \text{step} positive/negative
  
  \[
  \begin{align*}
  \text{range}(5) & : 0, 1, 2, 3, 4 \quad \text{(default from }= 0, \text{step }= 1) \\
  \text{range}(3, 8) & : 3, 4, 5, 6, 7 \quad \text{(default step }= 1) \\
  \text{range}(-2, 7, 3) & : -2, 1, 4 \\
  \text{range}(2, -5, -2) & : 2, 0, -2, -4
  \end{align*}
  \]
  
  (from and to can be any integer)

- Ranges are immutable, can be indexed like a list, sliced, and compared (i.e. generate the same numbers)

- \text{list} (\text{range} (...)) generates the explicit list of numbers

**Python shell**

```
> range(1, 10000000, 3)[2] 7
> range(1, 10000000, 3)[100:120:4]
range(301, 361, 12)
> range(1, 10000000, 3)[100:120:4][2:3]
range(325, 337, 12)
> list(range(5, 14, 3))
[5, 8, 11]
```
Question – What is \texttt{range(3,20,4)[2:4][1]}?

\begin{itemize}
\item[a)] 3
\item[b)] 7
\item[c)] 11
\item[d)] 15
\item[e)] 19
\item[f)] Don’t know
\end{itemize}
for - loop

- For every element in a sequence execute a block of code:

```python
for var in sequence:
    block
```

- Sequences can e.g. be lists, strings, ranges

- break and continue can be used like in a while-loop to break out of the for-loop or continue with the next element in the sequence

```python
Python shell
> for x in [1, "abc", [2, 3], 5.0]:
    print(x)
    1
    abc
    [2, 3]
    5.0
> for x in "abc":
    print(x)
    a
    b
    c
> for x in range(5, 15, 3):
    print(x)
    5
    8
    11
    14
```
Question – What is printed?

```python
Python shell
> for i in range(1, 4):
>     for j in range(i, 4):
>         print(i, j, sep=':', end=' ')  
```

a) 1:1 1:2 1:3 2:1 2:2 2:3 3:1 3:2 3:3

b) 1:1 1:2 1:3 2:2 2:3 3:3

c) 1:1 2:1 3:1 1:2 2:2 3:2 1:3 2:3 3:3

d) 1:1 2:1 3:1 2:2 3:2 3:3

e) Don’t know
Question – `break`, what is printed?

Python shell

```python
> for i in range(1, 4):
>     for j in range(1, 4):
>         print(i, j, sep=':', end=' ')
>         if j >= i:
>             break
```

a) `nothing`
b) `1:1`

![Smiley face] c) `1:1 2:1 2:2 3:1 3:2 3:3`
d) `1:1 2:2 3:3`
e) Don’t know

⚠️ In nested `for-` and `while-loops`, `break` only breaks the innermost loop
Palindromic substrings

- Find all **palindromic** substrings of length ≥ 2, i.e. substrings spelled identically forward and backwards:

  \[
  \text{abra}\_c\_d\_r\_a\_b\_r\_a\_t\_r\_a\_l\_l\_a\_l\_a
  \]

  \[
  i \quad j \quad i \quad j
  \]

- **Algorithm**: Test all possible substrings (brute force/exhaustive search)

- **Note**: the slice \( t[:::-1] \) is \( t \) reversed

```
palindrom.py

s = "abra\_c\_d\_r\_a\_b\_r\_a\_t\_r\_a\_l\_l\_a\_l\_a"

for i in range(len(s)):
    for j in range(i + 2, len(s) + 1):
        t = s[i:j]
        if t == t[:::-1]:
            print(t)
```

Python shell

```
|   aca   |
|   alla  |
|   allalla |
|    ll   |
|    lall |
|     lal |
|     alla |
|      ll |
```

Sieve of Eratosthenes

- Find all prime numbers \( \leq n \)

- Algorithm:

```
2 3 4 5 6 7 8 9 10 11 12 13 14 ... 2 3 4 5 6 7 8 9 10 11 12 13 14 ... 2 3 4 5 6 7 8 9 10 11 12 13 14 ... 2 3 4 5 6 7 8 9 10 11 12 13 14 ... 2 3 4 5 6 7 8 9 10 11 12 13 14 ...
```

```python
n = 100
prime = [True] * (n + 1)

for i in range(2, n):
    for j in range(2 * i, n + 1, i):
        prime[j] = False

for i in range(2, n + 1):
    if prime[i]:
        print(i, end=' ')```

| 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 | en.wikipedia.org/wiki/Sieve_of_Eratosthenes
while-else and for-else loops

- Both for- and while-loops can have an optional “else”:

  ```python
  for var in sequence:
    block
  else:
    block
  
  while condition:
    block
  else:
    block
  ```

- The “else” block is only executed if no `break` is performed in the loop

- The “else” construction for loops is specific to Python, and does not exist in e.g. C, C++ and Java
Linear search

**linear-search-while.py**

```python
L = [7, 3, 6, 4, 12, 'a', 8, 13]
x = 4

i = 0
while i < len(L):
    if L[i] == x:
        print(x, "at position", i, "in", L)
        break
    i = i + 1

if i >= len(L):
    print(x, "not in", L)
```

**linear-search-while-else.py**

```python
i = 0
while i < len(L):
    if L[i] == x:
        print(x, "at position", i, "in", L)
        break
    i = i + 1

if i >= len(L):
    print(x, "not in", L)
```

**linear-search-for.py**

```python
found = False
for i in range(len(L)):
    if L[i] == x:
        print(x, "at position", i, "in", L)
        found = True
        break

if not found:
    print(x, "not in", L)
```

**linear-search-for-else.py**

```python
for i in range(len(L)):
    if L[i] == x:
        print(x, "at position", i, "in", L)
        break
else:
    print(x, "not in", L)
```

**linear-search-built-in.py**

```python
if x in L:
    print(x, "at position", L.index(x), "in", L)
else:
    print(x, "not in", L)
```
Some performance considerations
String concatenation

- To concatenate two (or few) strings use
  
  \[ \text{str}_1 + \text{str}_2 \]

  \[ \text{var} += \text{str} \]

- To concatenate several/many strings use
  
  \[ '\text{.join}([\text{str}_1, \text{str}_2, \text{str}_3, \ldots, \text{str}_n]) \]

- Concatenating several strings by repeated use of + generates explicitly the longer-and-longer intermediate results; using join avoids this slowdown.
```python
from time import time
from matplotlib import pyplot as plt

ns = range(10_000, 1_000_000, 10_000)
time_string = []
time_list = []
for n in ns:
    start = time()
    s = ''
    for _ in range(n):
        s += 'abcdefgh'  # slow
    end = time()
    time_string.append(end - start)
    start = time()
    substrings = []
    for _ in range(n):
        substrings.append('abcdefgh')
        s = ''.join(substrings);
    end = time()
    time_list.append(end - start)

plt.plot(ns, time_string, label='string +=')
plt.plot(ns, time_list, label='''.join(list)''')
plt.xlabel('n')
plt.ylabel('time')
plt.legend()
plt.show()
```

≈ $2 \cdot 10^{-7} \cdot n$

≈ $4 \cdot 10^{-12} \cdot n^2$
The internal implementation of Python lists

- Accessing and updating list positions take the same time independently of position.
- Creating new / deleting entries in a list depends on position, Python optimizes towards updates at the end.
- Try to organize your usage of lists to insert / delete elements at the end. Use `L.append(element)` and `L.pop()`.
- Python lists internally have space for adding \( \approx 12.5\% \) additional entries at the end; when the reserved extra space is exhausted the list is moved to a new chunk of memory with \( \approx 12.5\% \) extra space.

```
memory

0   i   len(L) - 1

L[i:i] = [42] will move all trailing cells one position to right
move right
next cell to be used by L.append

github.com/python/cpython/blob/master/Objects/listobject.c
```
List insertions at begin vs end

Approximated times:
- Insertion at end: $\approx 2 \cdot 10^{-7} \cdot n$
- Append: $\approx 3 \cdot 10^{-7} \cdot n$
- Insertion at front: $\approx 7 \cdot 10^{-10} \cdot n^2$
Updates (insertions + deletions) in the middle of a list

```
list-updates.py

from time import time
from matplotlib import pyplot as plt

ns = range(0, 1_000_001, 10_000)
time_pos = []
L = list(range(1_000_000))  # L = [0, ..., 999_999]
for i in ns:
    start = time()
    for _ in range(1000):
        L[i:i] = [42]  # insert element before L[i]
        del L[i]  # remove L[i] from L
    end = time()
    time_pos.append(end - start)
plt.plot(ns, time_pos)
plt.xlabel('position')
plt.ylabel('time')
plt.show()
```
Tuples and lists

- tuples
- lists
- mutability
- list comprehension
- for-if, for-for
- list()
- any(), all()
- enumerate(), zip()
Tuples

(value₁, value₂, ..., valueₖ)

- Tuples can contain a sequence of zero or more elements, enclosed by "()"
- Tuples are immutable
- Tuple of length 0: ()
- Tuple of length 1: (value, )
  Note the comma to make a tuple of length one distinctive from an expression in parenthesis
- In many contexts a tuple with ≥ 1 elements can be written without parenthesis
- Accessors to lists also apply to tuples, slices, ...

Python shell

```python
> (1, 2, 3)
| (1, 2, 3)
> ()
| ()
> (42)
| 42
> (42,)
| (42,)
> 1, 2
| (1, 2)
> 42,
| (42,)
> x = (3, 7)
> x
| (3, 7)
> x = 4, 6
> x
| (4, 6)
> x[1] = 42
| TypeError: 'tuple' object does not support item assignment
```
Question – What value is \(((42,))\)?

a) 42  
b) (42)  
c) (42,)  
d) ((42,)),  
e) Don’t know
Question – What is x?

\[ x = [1, [2, 3], (4, 5)] \]
\[ x[2][0] = 42 \]

a) \[ [1, [42, 3], (4, 5)] \]
b) \[ [1, [2, 3], (42, 5)] \]
c) \[ [1, [2, 3], 42] \]
d) TypeError

e) Don’t know
Question – What tree is \(( 'A' , ( ( 'B' , 'C' ) , 'D' ) ) \)?

- a)
- b)
- c)
- d)
- e)
- f) Don’t know
Tuple assignment

- Parallel assignments

\[ x, y, z = a, b, c \]

is a short hand for a tuple assignment (right side is a single tuple)

\[ (x, y, z) = (a, b, c) \]

- First the right-hand side is evaluated completely, and then the individual values of the tuple are assigned to \( x, y, z \) left-to-right (length must be equal on both sides)
Nested tuple/lists assignments

- Let hand side can be nested (great for unpacking data)

\[
(x, (y, (a[0], w)), a[1]) = 1, (2, (3, 4)), 5
\]

- [...] and (...) on left side matches both lists and tuples of equal length (but likely you would like to be consistent with type of parenthesis)

```python
Python shell
>>> two_points = [(10, 25), (30, 40)]
>>> (x1, y1, x2, y2) = two_points
  ValueError: not enough values to unpack (expected 4, got 2)
>>> ((x1, y1), (x2, y2)) = two_points
>>> a = [None, None]
>>> v = ((2, (3, 4)), 5)
>>> ((y, (a[0], w)), a[1]) = v
>>> a
  [3, 5]
>>> [x, y, z] = (3, 5, 7)
>>> (x, y, z) = [3, 5, 7]
>>> [x, (y, z), w] = (1, [2, 3], 4)
>>> [x, (y, z), w] = (1, [2, (5, 6)], 4)
>>> z
  [5, 6]
```
Tuples vs lists: \( a \ += \ b \)

- **Lists**
  
  Extends existing list, i.e. same as \( a \ . \text{extend} \ (b) \)

- **Tuples**
  
  Must create a new tuple \( a \ + \ b \) and assign to \( a \) (since tuples are immutable)
*variable assignment*

- For a tuple of variable length a single *variable name* on the left side will be assigned a list of the remaining elements not matched by variables preceding/following *

- Example
  
  \[ a, *b, c = t \]

  is equivalent to

  \[ a = t[0] \]
  \[ b = t[1:-1] \]
  \[ c = t[-1] \]

- There can be a single * in a left-hand-side tuple (but one new * in each nested tuple)

---

**Python shell**

```python
(a,*b,c,d) = (1,2,3,4,5,6)
b | [2, 3, 4]
(a,*b,c,d) = (1,2,3)
b | []
(a,*b,c,d) = (1,2)
| ValueError: not enough values to unpack (expected at least 3, got 2)
v = ((1,2,3),4,5,6,(7,8,9,10))
((a,*b),*c,(d,*e)) = v
b | [2, 3]
c | [4, 5, 6]
e | [8, 9, 10]
```
Question – What is $b$?

\[ (*a, (b,), c) = ((1, 2), ((3, 4)), ((5,)), (6)) \]

a) $(1, 2)$
b) $(3, 4)$
c) $5$
d) $(5,)$
e) $(6)$
f) Don’t know
List comprehension (cool stuff)

- **Example:**
  ```python
  [ x*x for x in [1, 2, 3]]
  returns
  [1, 4, 9]
  ```

- **General**
  ```python
  [expression for variable in sequence]
  ```
  returns a list, where *expression* is computed for each element in *sequence* assigned to *variable*
List comprehension (more cool stuff)

- Similarly to the left-hand-side in assignments, the variable part can be a (nested) tuple of variables for unpacking elements:

\[
\text{[expression for tuple of variables in sequence]}
\]

**Python shell**

```
points = [(3, 4), (2, 5), (4, 7)]
[(x, y, x*y) for (x, y) in points]
[(3, 4, 12), (2, 5, 10), (4, 7, 28)]
[(x, y, x*y) for x, y in points]
[(3, 4, 12), (2, 5, 10), (4, 7, 28)]
[x, y, x*y for (x, y) in points]
SyntaxError: invalid syntax
```
List comprehension – for-if and multiple for

- List comprehensions can have nested for-loops

\[
\text{[expression for } v_1 \text{ in } s_1 \text{ for } v_2 \text{ in } s_2 \text{ for } v_3 \text{ in } s_3]\]

- Can select a subset of the elements by adding an if-condition

\[
\text{[expression for } v_1 \text{ in } s_1 \text{ if } \text{condition}]\
\]

- and be combined...

Python shell

```python
> [(x, y) for x in range(1, 3) for y in range(4, 6)]
| [(1, 4), (1, 5), (2, 4), (2, 5)]
> [x for x in (1, 2) for x in (4, 5)]
| [4, 5, 4, 5]
> [x for x in range(1, 101) if x % 7 == 1 and x % 5 == 2]
| [22, 57, 92]
> [(x, y, x*y) for x in range(1, 11) if 6 <= x <= 7 for y in range(x, 11) if 6 <= y <= 7 and not x == y]
| [(6, 7, 42)]
```
points = [(3,7), (4,10),(12,3), (9,11), (7,5)]
print([(x, y) for x, y in points if x < y])

a) print([x, y for x, y in points if x < y])
b) print([(x, y) for p in points if p[0] < p[1]])
c) print([p for p in points if p[0] < p[1]])
d) print([[x, y] for x, y in points if x < y])
e) Don’t know
any, all

- **any(L)** checks if at least one element in the sequence L is true (list, sequence, strings, ranges, ...)
  
  ```python
  any([[False, True, False]])
  ```

- **all(L)** checks if all elements in the sequence L are true
  
  ```python
  all([[False, False, True]])
  ```

- **any and all** returns True or False
  
  ```python
  L = (7, 42, 13)
  any([x == 42 for x in L])
  all([x == 42 for x in L])
  ```

---

**Python shell**

```
> any((False, True, False))
| True
> any([[False, False, False]])
| False
> any([])
| False
> all([[False, False, True]])
| False
> all((True, True, True))
| True
> all([])
| True
> L = (7, 42, 13)
> any([x == 42 for x in L])
| True
> all([x == 42 for x in L])
| False
```
enumerate

\[
\text{list(enumerate(L))}
\]
returns
\[
[(0,L[0]), (1,L[1]), \ldots, (\text{len}(L)-1,L[-1])]\]

Python shell

```python
>>> points = [(1,2),(3,4),(5,6)]
>>> [(idx, x*y) for idx, (x, y) in enumerate(points)]
[(0, 2), (1, 12), (2, 30)]
>>> L = ('a','b','c')
>>> list(enumerate(L))
[(0, 'a'), (1, 'b'), (2, 'c')]
>>> L_ = []
>>> for idx in range(len(L)):
>>>     L_.append((idx, L[idx]))
>>> print(L_)
[(0, 'a'), (1, 'b'), (2, 'c')]
>>> list(enumerate(['a', 'b', 'c'], start=7))
[(7, 'a'), (8, 'b'), (9, 'c')]
```
zip

\[
\text{list(zip}(L_1, L_2, \ldots, L_k) \)) = [(L_1[0], L_2[0], \ldots, L_k[0]), \ldots, (L_1[n], L_2[n], \ldots, L_k[n])] \]

where \( n = \min(\text{len}(L_1), \text{len}(L_2), \ldots, \text{len}(L_k)) \)

- Example (“matrix transpose”):
  \[
  \text{list(zip}([1, 2, 3], [4, 5, 6], [7, 8, 9]))
  \]
  returns
  \[
  [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
  \]

<table>
<thead>
<tr>
<th>Python shell</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = [1, 2, 3]</td>
</tr>
<tr>
<td>y = [4, 5, 6]</td>
</tr>
<tr>
<td>zip(x, y)</td>
</tr>
<tr>
<td>&lt;zip at 0xb02b530&gt;</td>
</tr>
<tr>
<td>points = list(zip(x, y))</td>
</tr>
<tr>
<td>print(points)</td>
</tr>
<tr>
<td>[(1, 4), (2, 5), (3, 6)]</td>
</tr>
</tbody>
</table>
first = ['Donald', 'Mickey', 'Scrooge']
last = ['Duck', 'Mouse', 'McDuck']

for i, (a, b) in enumerate(zip(first, last), start=1):
    print(i, a, b)

1 Donald Duck
2 Mickey Mouse
3 Scrooge McDuck
(Simple) functions

- You can define your own functions using:

  ```python
def function-name (var_1, ..., var_k):
    body code
  ```

- If the body code executes

  ```python
  return expression
  ```

  the result of expression will be returned by the function. If expression is omitted or the body code terminates without performing return, then None is returned.

- When calling a function name (value_1, ..., value_k) body code is executed with var_i=value_i

<table>
<thead>
<tr>
<th>Python shell</th>
</tr>
</thead>
</table>
| > def sum3(x, y, z):
  |   return x + y + z |
| > sum3(1, 2, 3) |
|  | 6 |
| > sum3(5, 7, 9) |
|  | 21 |
| > def powers(L, power):
  |   P = [x**power for x in L]
  |   return P |
| > powers([2, 3, 4], 3) |
|  | [8, 27, 64] |
def even(x):
    if x % 2 == 0:
        return True
    else:
        return False

print((even(7), even(6)))

a) (False, False)
b) (False, True)
c) (True, False)
d) (True, True)
e) Don’t know
Geometric orientation test

Purpose of example

- illustrate tuples
- list comprehension
- matplotlib.pyplot
- floats are strange

\[
\text{det} = \begin{vmatrix} 1 & q_x & q_y \\ 1 & r_x & r_y \\ 1 & p_x & p_y \end{vmatrix} = r_x p_y - p_x r_y - q_x p_y + p_x q_y + q_x r_y - r_x q_y
\]

\( \text{det} > 0 \) 180° < α < 360°

\( 0° < \alpha < 180° \)

(0.5 + ε, 0.5 + ε)

(0.5, 0.5)

q = (12, 12)

r = (24, 24)

Kettner, Mehlhorn, Pion, Schirra, Yap:
Classroom Examples of Robustness Problems in Geometric Computations
import matplotlib.pyplot as plt

N = 256
delta = 1 / 2**54
q = (12, 12)
r = (24, 24)
P = []  # points (i, j, det)

for i in range(N):
    for j in range(N):
        p = (1/2 + i * delta, 1/2 + j * delta)
        det = (q[0]*r[1] + r[0]*p[1] + p[0]*q[1] - r[0]*q[1] - p[0]*r[1] - q[0]*p[1])
        P.append((i, j, det))

pos = [(i, j) for i, j, det in P if det > 0]
eg = [(i, j) for i, j, det in P if det < 0]
zero = [(i, j) for i, j, det in P if det == 0]

plt.subplot(facecolor='lightgrey', aspect='equal')
plt.xlabel('i')
plt.ylabel('j', rotation=0)

for points, color in [(pos, "b"), (neg, "r"), (zero, "y")]:
    X = [x for x, y in points]
    Y = [y for x, y in points]
    plt.plot(X, Y, color + ".")

plt.plot([-1, N], [-1, N], "k-")
plt.show()
Dictionaries and Sets

- dict
- set
- frozenset
- set/dict comprehensions
Dictionaries (type dict)

\[
\{ \text{key}_1 : \text{value}_1, \ldots, \text{key}_k : \text{value}_k \}
\]

- Stores a mutable set of (key, value) pairs, denoted items, with distinct keys, i.e. maps keys to values.
- Constructing empty dictionary: `dict()` or `{}`
- `dict[key]` lookup for key in dictionary, and returns associated value. Key must be present otherwise a KeyError is raised.
- `dict[key] = value` assigns value to key, overriding existing value if present.

```
{ 'a': 7, 'foo': '42nd', 5: 29, '5': 44, 5.5: False, False: True, (3, 4): 'abc' }
```

Distinct keys, i.e. not "=="

https://docs.python.org/3/tutorial/datastructures.html#dictionaries
Dictionaries (type dict)

Python shell

```python
> d = {'a': 42, 'b': 57}
> d
| {'a': 42, 'b': 57}

> d.keys()
| dict_keys(['a', 'b'])

> list(d.keys())
| ['a', 'b']

> d.items()
| dict_items([('a', 42), ('b', 57)])

> list(d.items())
| [('a', 42), ('b', 57)]

> for key in d:
|     print(key)
| a
| b

> for key, val in d.items():
|     print("Key", key, ", "has value", val)
| Key a has value 42
| Key b has value 57

> {5: 'a', 5.0: 'b'}
| {5: 'b'}
```

Python shell

```python
> surname = dict(zip(['Donald', 'Mickey', 'Scrooge'], ['Duck', 'Mouse', 'McDuck']))
> surname['Mickey']
| 'Mouse'
```
Dictionaries (type dict)

Python shell

```python
> gradings = [('A', 7), ('B', 4), ('A', 12), ('C', 10), ('A', 7)]
> grades = {}  # empty dictionary
> for student, grade in gradings:
>     if student not in grades:
>         grades[student] = []
>         grades[student].append(grade)
> grades
| {'A': [7, 12, 7], 'B': [4], 'C': [10]}
> print(grades['A'])
| [7, 12, 7]
> print(grades['E'])  # can only lookup keys in dictionary
| KeyError: 'E'
> print(grades.get('E'))  # .get returns None if key not in dictionary
| None
> print(grades.get('E', []))  # change default return value
| []
> print(grades.get('A', []))
| [7, 12, 7]
```
<table>
<thead>
<tr>
<th>Dictionary operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>len(d)</td>
<td>Items in dictionary</td>
</tr>
<tr>
<td>d[key]</td>
<td>Lookup key</td>
</tr>
<tr>
<td>d[key] = value</td>
<td>Update value of key</td>
</tr>
<tr>
<td>del d[key]</td>
<td>Delete an existing key</td>
</tr>
<tr>
<td>key in d</td>
<td>Key membership</td>
</tr>
<tr>
<td>key not in d</td>
<td>Key non-membership</td>
</tr>
<tr>
<td>clear()</td>
<td>Remove all items</td>
</tr>
<tr>
<td>copy()</td>
<td>Shallow copy</td>
</tr>
<tr>
<td>get(key), get(key, default)</td>
<td>d[key] if key in dictionary, otherwise None or default</td>
</tr>
<tr>
<td>items()</td>
<td>View of the dictionaries items</td>
</tr>
<tr>
<td>keys()</td>
<td>View of the dictionaries keys</td>
</tr>
<tr>
<td>values()</td>
<td>View of the dictionaries values</td>
</tr>
<tr>
<td>pop(key)</td>
<td>Remove key and return previous value</td>
</tr>
<tr>
<td>popitem()</td>
<td>Remove and return an arbitrary item</td>
</tr>
<tr>
<td>update()</td>
<td>Update key/value pairs from another dictionary</td>
</tr>
</tbody>
</table>

https://docs.python.org/3/library/stdtypes.html#mapping-types-dict
Order returned by `list(d.keys())`?

---

**The Python Standard Library**
**Mapping Types — dict**

“Dictionaries preserve insertion order. Note that updating a key does not affect the order. Keys added after deletion are inserted at the end.” (since Python 3.7)

[docs.python.org/3/library/stdtypes.html](https://docs.python.org/3/library/stdtypes.html)

---

**Python shell**

```python
>>> d = {'d': 1, 'c': 2, 'b': 3, 'a':4}
>>> d['x'] = 5      # new key at end
>>> d['c'] = 6      # overwrite value
>>> del d['b']     # remove key 'b'
>>> d['b'] = 7     # reinsert key 'b' at end
>>> d
{'d': 1, 'c': 6, 'a': 4, 'x': 5, 'b': 7}
```

---

Raymond Hettinger @ Twitter

See also Raymond’s talk @ PyCon 2017

[Modern Python Dictionaries](https://docs.google.com/presentation/d/1yv9vz0a6z7uKUuq7vZCtGvzX9ZPlpurA/edit?usp=sharing)

A confluence of a dozen great ideas
Dictionary comprehension

- Similarly to creating a list using list comprehension, one can create a set of key-value pairs:

\[
\{\text{key : value for variable in list}\}
\]

Python shell

```
> names = ['Mickey', 'Donald', 'Scrooge']
> list(enumerate(names, start=1))
| [(1, 'Mickey'), (2, 'Donald'), (3, 'Scrooge')]
> dict(enumerate(names, start=1))
| {1: 'Mickey', 2: 'Donald', 3: 'Scrooge'}
> {name: idx for idx, name in enumerate(names, start=1)}
| {'Mickey': 1, 'Donald': 2, 'Scrooge': 3}
```
Sets (set and frozenset)

\{'value_1, \ldots, value_k\}

- Values of type set represent mutable sets, where "==" elements only appear once
- Do not support: indexing, slicing
- frozenset is an immutable version of set

<table>
<thead>
<tr>
<th>Python shell</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; S = {2, 5, 'a', 'c'}</td>
</tr>
<tr>
<td>&gt; T = {3, 4, 5, 'a'}</td>
</tr>
<tr>
<td>&gt; S</td>
</tr>
<tr>
<td>&gt; S &amp; T</td>
</tr>
<tr>
<td>&gt; S ^ T</td>
</tr>
<tr>
<td>&gt; S - T</td>
</tr>
<tr>
<td>&gt; S &lt;= T</td>
</tr>
<tr>
<td>&gt; S &lt; T</td>
</tr>
<tr>
<td>&gt; S &gt;= T</td>
</tr>
<tr>
<td>&gt; S &gt; T</td>
</tr>
<tr>
<td>&gt; len(S)</td>
</tr>
<tr>
<td>&gt; S.add(x)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>T</td>
</tr>
<tr>
<td>S &amp; T</td>
<td>Set intersection</td>
</tr>
<tr>
<td>S – T</td>
<td>Set difference</td>
</tr>
<tr>
<td>S ^ T</td>
<td>Symmetric difference</td>
</tr>
<tr>
<td>set()</td>
<td>Empty set</td>
</tr>
<tr>
<td>set(L)</td>
<td>Create set from list</td>
</tr>
<tr>
<td>x in S</td>
<td>Membership</td>
</tr>
<tr>
<td>x not in S</td>
<td>Non-membership</td>
</tr>
<tr>
<td>S.isdisjoint(T)</td>
<td>Disjoint sets</td>
</tr>
<tr>
<td>S &lt;= T</td>
<td>Subset</td>
</tr>
<tr>
<td>S &lt; T</td>
<td>Proper subset</td>
</tr>
<tr>
<td>S &gt;= T</td>
<td>Superset</td>
</tr>
<tr>
<td>S &gt; T</td>
<td>Proper superset</td>
</tr>
<tr>
<td>len(S)</td>
<td>Size of S</td>
</tr>
<tr>
<td>S.add(x)</td>
<td>Add x to S (not frozenset)</td>
</tr>
</tbody>
</table>

https://docs.python.org/3/tutorial/datastructures.html#sets
https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset
Question – What value has the expression?

```
sorted({ 5, 5.5, 5.0, '5' })
```

a) {'5', 5, 5.0, 5.5}
b) {5, 5.5}
c) ['5', 5, 5.0, 5.5]
d) ['5', 5, 5.5]
e) TypeError
f) Don’t know
Set comprehension

- Similarly to creating a list using list comprehension, one can create a set of values (also using nested for- and if-statements):

\[
\{\text{value for variable in list}\}
\]

- A value occurring multiple times as value will only be included once

```python
primes_set.py
n = 101
not_primes = {m for f in range(2, n) for m in range(2 * f, n, f)}
primes = set(range(2, n)) - not_primes
```

```bash
Python shell
> L = ['a', 'b', 'c']
> {(x, (y, z)) for x in L for y in L for z in L if x != y and y != z and z != x}
  | {('a', ('b', 'c')), ('a', ('c', 'b')), ('b', ('a', 'c')), ..., ('c', ('b', 'a'))}
```
Hash, equality, and immutability

- Keys for dictionaries and sets must be *hashable*, i.e. have a `__hash__()` method returning an integer that does not change over their lifetime and an `__eq__()` method to check for equality with “==”.

  ```python
  'abc'.__hash__()  # could e.g. return 624337162
  (624337162).__hash__()  # would also return 624337162
  ```

- All built-in immutable types are hashable. In particular tuples of immutable values are hashable. I.e. trees represented by nested tuples like `((('a'), 'b'), ('c', ('d', 'e')))` can be used as dictionary keys or stored in a set.
Sketch of internal set implementation

```plaintext
hash('abc') % size

elements with same hash value (hopefully few) search linearly using ==

Raymond Hettinger, Modern Python Dictionaries - A confluence of a dozen great ideas
```
Module \texttt{collections} (container datatypes)

- Python builtin containers for data: \texttt{list, tuple, dict, and set.}
- The module \texttt{collections} provides further alternatives (but these are not part of the language like the builtin containers)

\begin{itemize}
  \item \texttt{deque} \quad \texttt{double ended queue}
  \item \texttt{namedtuple} \quad \texttt{tuples allowing access to fields by name}
  \item \texttt{counter} \quad \texttt{special dictionary to count occurrences of elements}
  \item ...
\end{itemize}

\url{https://docs.python.org/3/library/collections.html}
deque – double ended queues

- Extends lists with efficient updates at the front
- Inserting at the front of a standard Python list takes linear time in the size of the list – very slow for long lists

### Python shell

```python
from collections import deque

L = list()  
L.append(1)  
L.append(2)  
L.insert(0, 0)  # insert at the front  
L.insert(0, -1)  # slow for long lists  
L.insert(0, -2)  
L  
-2, -1, 0, 1, 2

d = deque()  # create empty deque  
d.append(1)  
d.append(2)  
d.appendleft(0)  # efficient  
d.appendleft(-1)  
d.appendleft(-2)  
d  
deque([-2, -1, 0, 1, 2])  
for e in d: print(e, end=' ', ward)  
-2, -1, 0, 1, 2,
```
namedtuple – tuples with field names

- Compromise between tuple and dict, can increase code readability

```
Python shell

> person = ('Donald Duck', 1934, '3 feet')  # as tuple
> person[1]  # not clear what is accessed
| 1934
> person = {'name': 'Donald Duck', 'appeared': 1934, 'height': '3 feet'}  # as dict
> person['appeared']  # more clear what is accessed, but ['...'] overhead
| 1934
> from collections import namedtuple
> Person = namedtuple('Person', ['name', 'appeared', 'height'])  # create new type
> person = Person('Donald Duck', 1934, '3 feet')  # as namedtuple
> person  # short and clear
| Person(name='Donald Duck', appeared=1934, height='3 feet')
> person.appeared  # short and clear
| 1934
> person[1]  # still possible
| 1934
```
**Counter** – dictionaries for counting

```python
from collections import Counter

text = 'abracadabra'

fq = Counter(text)  # create new counter from a sequence
print(fq)  # Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})  # frequencies of the letters

print(fq['a'])  # 5

print(fq.most_common(3))  # [('a', 5), ('b', 2), ('r', 2)]

fq['x'] += 5  # increase count of 'x', also valid if 'x' not in Counter yet

print(Counter('aaabbbcc') - Counter('aabdd'))  # counters can be subtracted
# Counter({'b': 2, 'c': 2, 'a': 1})

print(Counter([1, 2, 1, 3, 4, 5]) + Counter([3, 3, 3]))  # counters can be added
# Counter({3: 4, 1: 2, 2: 1, 4: 1, 5: 1})
```
Handin 3 & 4 – Triplet distance (Dobson, 1975)

How similar are the trees?

(a) $(((A', F'), B'), (D', (C', E')))$  
(b) $(((D', A'), B'), (F'), (C', E'))$

arxiv.org/abs/1706.10284
Handin 3 & 4 – Triplet distance (Dobson, 1975)

Consider all $\binom{n}{3}$ subsets of size three, and count how many do not have identical substructure (topology) in the two trees.

\begin{itemize}
\item \text{(a)} \quad \\
\item \text{(b)}
\end{itemize}

\begin{itemize}
\item \text{canonical ordering}
\end{itemize}

arxiv.org/abs/1706.10284
Functions

- functions
- return
- scoping
- arguments
- keyword arguments
- *, **
- global variables
(Simple) functions

- You can define your own functions using:

  ```
  def function-name (var₁, ..., varₖ):
  body code
  ```

  *var₁, ..., varₖ* are the formal parameters

- If the body code executes

  ```
  return expression
  ```

  the result of *expression* will be returned by the function. If expression is omitted or the body code terminates without performing `return`, then `None` is returned.

- When calling a function `name (value₁, ..., valueₖ)` body code is executed with `varᵢ=valueᵢ`
Questions – \texttt{poly(3, "10", '3')} ?

\begin{verbatim}
def poly(z, x, y):
    return z*x + y
\end{verbatim}

\begin{itemize}
\item [a)] 33
\item [b)] 1010103
\item [c)] '33'
\item [d)] '1010103'
\item [e)] TypeError
\item [f)] Don’t know
\end{itemize}
Why functions?

- Avoid writing the same code multiple times, *re-usability*
- Be able to *name a functionality*
- Clearly state the functionality of a piece of code, *abstraction*: *Input* = arguments, *output* = return value (and/or side effects)
- *Encapsulate* code with clear interface to the dependency to the outside world/code
- Share functionality in modules/libraries/packages with other users, *code sharing*
- Increase *readability* of code, smaller independent blocks of code
- Easier systematically *testing* of code
- ...

---

Some other Python language features helping structuring programs
- Object orientation
- Modules
- Decorators
- Context managers
- Exceptions
- Doc strings
- doctest
Local variables in functions

- The formal arguments and variables assigned to in the body of a function are created as temporary local variables.

<table>
<thead>
<tr>
<th>Global variables</th>
<th>Local variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum3 &lt;function&gt;</td>
<td>x 4</td>
</tr>
<tr>
<td>a 3</td>
<td>y 5</td>
</tr>
<tr>
<td>y 42</td>
<td>z 6</td>
</tr>
<tr>
<td></td>
<td>a 9</td>
</tr>
</tbody>
</table>
|                           | b 15

Python shell

```python
> def sum3(x, y, z):
>     a = x + y
>     b = a + z
>     return b
> a = 3
> y = 42
> w = sum3(4, 5, 6)
> w
| 15
> a
| 3
> b
| NameError: name 'b' is not defined
> x
| NameError: name 'x' is not defined
> y
| 42
> sum3
| <function sum3 at 0x0356DA98>
```

state just before return b
Global variables

- Variables in function bodies that are only read, are considered access to *global variables*

---

**Python shell**

```python
> prefix = "The value is"
> def nice_print(x):
>     print(prefix, x)
> nice_print(7)
| The value is 7
> prefix = "Value ="
> nice_print(42)
| Value = 42
```

<table>
<thead>
<tr>
<th>Global variables</th>
<th>Local variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>nice_print    &lt;function&gt;</td>
<td>x</td>
</tr>
<tr>
<td>prefix         &quot;Value =&quot;</td>
<td>42</td>
</tr>
</tbody>
</table>

State just before returning from *2nd nice_print*
Global variables that should be updated in the function body must be declared global in the body:

```
global variable, variable, ...
```

Note: If you only need to read a global variable, it is not required to be declared global (but would be polite to the readers of your code)

```python
> counter = 1
> def counted_print(x):
>     global counter
>     print("(%d)" % counter, x)
>     counter += 1
> counted_print(7)
| (1) 7
> counted_print(42)
| (2) 42
> def counted_print(x):
>     print("(%d)" % counter, x)
>     counter += 1
> counted_print(7)
| UnboundLocalError: local variable 'counter' referenced before assignment
```

Since `counter` assigned in body, `counter` will be considered to be a local variable
Question – What value is printed?

```python
x = 1

def f(a):
    global x
    x = x + 1
    return a + x

print(f(2) + f(4))
```

a) 6  
b) 7  
c) 8  
d) 9  
e) 10  
f) 11  
g) 12  
h) Don’t know
Arbitrary number of arguments

- If you would like your function to be able to take a variable number of additional arguments in addition to the required, add a `*variable` as the last argument.

- In a function call `variable` will be assigned a tuple with all the additional arguments.

```python
Python shell
> def my_print(x, y, *L):
    print("x =", x)
    print("y =", y)
    print("L =", L)
> my_print(2, 3, 4, 5, 6, 7)
  x = 2
  y = 3
  L = (4, 5, 6, 7)
> my_print(42)
TypeError: my_print() missing 1 required positional argument: 'y'
```
Unpacking a list of arguments in a function call

- If you have list $L$ (or tuple) containing the arguments to a *function call*, you can unpack them in the function call using $*L$

  $L = [x, y, z]$
  
  $f(*L)$

  is equivalent to calling

  $f(L[0], L[1], L[2])$

  i.e.

  $f(x, y, z)$

- Note that $f(L)$ would pass a single argument to $f$, namely a list

- In a function call several $*$ expressions can appear, e.g. $f(*L_1, x, *L_2, *L_3)$
```python
import math

def norm(x, y):
    return math.sqrt(x * x + y * y)

norm(3, 5)  # 5.830951894845301
point = (3, 4)
print(*point, sep=':')  # 3:4
norm(point)  # TypeError: norm() missing 1 required positional argument: 'y'

def dist(x0, y0, x1, y1):
    return math.sqrt((x1 - x0) ** 2 + (y1 - y0) ** 2)

p = 3, 7
q = 7, 4
dist(p, q)  # TypeError: dist() missing 2 required positional arguments: 'x1' and 'y1'
dist(*p, *q)  # 5.0
```
Question – How many arguments should \( f \) take?

\[
a = [1, 2, 3] \\
b = [4, 5] \\
c = (6, 7, 8) \\
d = (9, 10) \\
f(*a, b, c, *d)
\]

a) 4 

b) 5 

c) 6 

\[\text{Correct Answer} \] 

d) 7 

e) 8 

f) 9 

g) 10 

h) Don’t know
```python
L = [[1, 2, 3], [4, 5], [6, 7, 8]]

a) [((1, 2, 3),), ((4, 5),), ((6, 7, 8),)]
b) [((1, 4, 6),), ((2, 5, 7),)]
c) [(1, 2), (4, 5), (6, 7)]
d) [(1, 2, 3), (4, 5), (6, 7, 8)]
e) [((1, 2, 3), [4, 5], [6, 7, 8])]  
f) Don’t know
```

**Python shell**
```
> list(zip((1, 2, 3), (4, 5, 6)))
[(1, 4), (2, 5), (3, 6)]
```
Keyword arguments

- Previously we have seen the following (strange) function calls

  ```python
  print(7, 14, 15, sep=":", end="")
  enumerate(my_list, start=1)
  ```

- `name =` refers to one of the formal arguments, known as a **keyword argument**. A `name` can appear at most once in a function call.

- In function calls, keyword arguments must follow positional arguments.

- Can e.g. be useful if there are many arguments, and the order is not obvious, i.e. improves readability of code:

  ```python
  complicated_function(
      name = "Mickey",
      city = "Duckburg",
      state = "Calisota",
      occupation = "Detective",
      gender = "Male")
  ```

Python shell

```
> def sub(x, y):
    return x - y
> sub(9, 4)
5
> sub(y=9, x=4)
-5
```
Keyword arguments, default values

- When calling a function arguments can be ommitted if the corresponding arguments in the function definition have default values `argument = value`.

```python
Python shell
> def my_print(a, b, c=5, d=7):
    print("a=%s, b=%s, c=%s, d=%s" % (a, b, c, d))
> my_print(2, d=3, b=4)
  a=2, b=4, c=5, d=3
```
Question – What is \( f(6, z=2) \)?

```python
def f(x, y=3, z=7):
    return x + y + z
```

a) 10

b) 11

c) 16

d) `TypeError: f() missing 1 required positional argument: 'y'`

e) Don’t know
Keyword arguments, mutable default values

- Be carefull: Default value will be shared among calls (which can be usefull)

---

**The Python Language Reference**

8.6 Function definitions

"Default parameter values are evaluated from left to right when the function definition is executed. This means that the expression is evaluated once, when the function is defined, and that the same “pre-computed” value is used for each call. This is especially important to understand when a default parameter is a mutable object, such as a list or a dictionary: if the function modifies the object (e.g. by appending an item to a list), the default value is in effect modified. This is generally not what was intended. A way around this is to use None as the default"
Function call, dictionary of keyword arguments

- If you happen to have a *dictionary* containing the keyword arguments you want to pass to function, you can give all dictionary items as arguments using the single argument `**dictionary`.

```
Python shell
> print(3, 4, 5, sep=":", end='\n')
| 3:4:5#
> print_kwarg = {'sep': ':', 'end': '\n'}
> print(3, 4, 5, **print_kwarg)
| 3:4:5#
```
Function definition, arbitrary keyword arguments

- If you want a function to accept arbitrary keyword arguments, add an argument `**argument` to the function definition.
- When the function is called `argument` will be assigned a dictionary containing the excess keyword arguments.

```python
> my_print(x=27, y=42, a=7)
a = 7
b = 3
c = {'x': 27, 'y': 42}
```
Example

<table>
<thead>
<tr>
<th>Python shell</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; L1 = [1, 'a']</td>
</tr>
<tr>
<td>&gt; L2 = ['b', 2, 3]</td>
</tr>
<tr>
<td>&gt; D1 = {'y':4, 's':10}</td>
</tr>
<tr>
<td>&gt; D2 = {'t':11, 'z':5.0}</td>
</tr>
</tbody>
</table>
| > def f(a, b, c, d, e, *f, q=0, x=1, y=2, z=3, **kw):
|   print("a=%s, b=%s, c=%s, d=%s, e=%s, " % (a, b, c, d, e),
|   "f=%s\n" % str(f),
|   "q=%s, x=%s, y=%s, z=%s, " % (q, x, y, z),
|   "kw=%s" % kw,
|   sep="") |
| > f(7, *L1, 9, *L2, x=7, **D1, w=42, **D2) |
|   a=7, b=1, c=a, d=9, e=b, f=(2, 3) |
|   q=0, x=7, y=4, z=5.0, kw={'w': 42, 's': 10, 't': 11} |

non-keyword arguments must appear before keyword arguments
Forwarding function arguments

- * and ** can e.g. be used to forward (unknown) arguments to other function calls

```python
> def my_print(*positional_arguments, sep=":", **keyword_arguments):
    print(*positional_arguments, sep=sep, **keyword_arguments)
> my_print(7, 42)
  7:42
> my_print("x", "y", end="<")
  x:y<
> my_print("x", "y", sep="_")
  x_y
```
Local function definitions and namespaces

- Function definitions can contain (nested) local function definitions, only accessible inside the function
- *static/lexical scoping*, i.e. can see from the code which variables are in scope

```python
Python shell
> def a(x):
    def b(y):
        print("b: y=%s x=%s" % (y, x))
        c(y + 1)
    def c(z):
        print("c: z=%s x=%s" % (z, x))
        print("a: x=%s" % x)
    b(x + 1)
> a(42)
  | a: x=42
  | b: y=43 x=42
  | c: z=44 x=42
```

![Diagram](https://via.placeholder.com/150)
Example – nested function definitions

Python shell

> def a(x):
    
def b(y):
        print("Enter b (y=%s, x=%s)" % (y, x))
    
c(y + 1)
    print("leaving b")
    
def c(x):  # x hides argument of function a
        def d(z):
            print("Enter d (z=%s, x=%s)" % (z, x))
            print("leaving d")
        
        print("Enter c (x=%s)" % x)
        d(x + 1)
        print("leaving c")
        print("Enter a (x=%s)" % x)
        b(x + 1)
        print("leaving a")

> a(5)
| Enter a (x=5)
| Enter b (y=6, x=5)
| Enter c (x=7)
| Enter d (z=8, x=7)
| leaving d
| leaving c
| leaving b
| leaving a
Example – nested functions and default values

```python
# Python shell

> def init_none(var_name):
>     print('initializing', var_name)
>     return None  # redundant line

> def f(a=init_none('a')):
>     def g(b=init_none('b')):
>         print('b =', b)
>         print("a =", a)
>         g(a + 1)
>     initializing a

> f(10)
> initializing b  !
> a = 10
> b = 11
```
The `nonlocal` statement causes the listed identifiers to refer to previously bound variables in the nearest *enclosing scope excluding globals*.

- `nonlocal variable, variable, ...`

---

**Python shell**

```python
x = 0
def f():
    y = 1
def f_helper(z):
    global x
    nonlocal y
    print("(%s:%s) %s" % (x, y, z))
    y += 1
    x += 3
    f_helper(7)
f_helper(42)
f()
| (0:1) 7
| (3:2) 42
f()
| (6:1) 7
| (9:2) 42
```

[docs.python.org/3/reference/simple_stmts.html#the-nonlocal-statement](https://docs.python.org/3/reference/simple_stmts.html#the-nonlocal-statement)
A note on Python and functions

- Similarities between Python and other languages:
  - functions are widely supported (sometimes called methods and procedures)
  - scoping rules is present in many languages (but details differ)

- The following are very Python specific (but nice):
  - how to handle global, local and nonlocal variables
  - keyword arguments
    - *, **
Recursion

- symbol table
- stack frames
Recursion

Recursive function
≡
"function that calls itself"

```python
def recursive_function(x):
    if x > 0:
        print("start", x)
        recursive_function(x - 1)
        print("end", x)
    else:
        print("done")

recursive_function(5)
```

```
start 5
  recursive_function(5)
  start 4
  recursive_function(4)
  start 3
  recursive_function(3)
  start 2
  recursive_function(2)
  start 1
  recursive_function(1)
  recursive_function(0)
  done
end 5
end 4
end 3
end 2
end 1
done
```

> recursive_function(5)
  start 5
  start 4
  start 3
  start 2
  start 1
  done
  end 1
  end 2
  end 3
  end 4
  end 5
```
Recursion

Recursion stack when $x = 0$ is reached

```
> def recursive_function(x):
>     if x > 0:
>         print("start", x)
>         recursive_function(x - 1)
>         print("end", x)
>     else:
>         print("done")
>
> recursive_function(5)
```

```
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
```python
def rec(x):
    if x > 0:
        print("start", x)
        rec(x - 1)
        rec(x - 1)
        print("end", x)
    else:
        print("done")
```
Question – How many times does `rec(5)` print ”done”?

Python shell

```
> def rec(x):
    if x > 0:
        print("start", x)
        rec(x - 1)
        rec(x - 1)
        rec(x - 1)
        print("end", x)
    else:
        print("done")
```

a) 3
b) 5
c) 15
d) 81
e) 125
f) 243
g) Don’t know
Factorial

\[ n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 \]

**Observation**

(_recursive definition)

1! = 1

n! = n \cdot (n - 1) !

---

**factorial.py**

```python
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)
```

**factorial_iterative.py**

```python
def factorial(n):
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result
```
Binomial coefficient \( \binom{n}{k} \)

- \( \binom{n}{k} = \) number of ways to pick \( k \) elements from a set of size \( n \)
- \[ \binom{n}{k} = \begin{cases} \frac{1}{k} + \frac{1}{k-1} & \text{if } k = 0 \text{ or } k = n \\ (n-1)^k + (n-1)^{k-1} & \text{otherwise} \end{cases} \]

**binomial_recursive.py**

```python
def binomial(n, k):
    if k == 0 or k == n:
        return 1
    return binomial(n - 1, k) + binomial(n - 1, k - 1)
```

- Unfolding computation shows \( \binom{n}{k} \) 1’s are added \( \rightarrow \) slow
Binomial coefficient \( \binom{n}{k} \)

Observation \( \binom{n}{k} = \frac{n!}{(n-k)! \cdot k!} \)

- Unfolding computation shows \( 2n - 2 \) multiplications and 2 divisions \( \rightarrow \text{fast} \)
- Intermediate value \( n! \) can have significantly more digits than result (bad)

```python
bionomial_factorial.py
def binomial(n, k):
    return factorial(n) // factorial(k) // factorial(n - k)
```
Binomial coefficient \( \binom{n}{k} \)

Observation
\[
\binom{n}{k} = \frac{n \cdot (n - 1) \cdot (n - 2) \cdots (n - k + 1)}{k \cdot (k - 1) \cdot (k - 2) \cdots 1} = \binom{n - 1}{k - 1} \cdot \frac{n}{k}
\]

**bionomial_recursive_product.py**

```python
def binomial(n, k):
    if k == 0:
        return 1
    else:
        return binomial(n - 1, k - 1) * n // k
```

- Unfolding computation shows 2k - 2 multiplications and k divisions \(\rightarrow\) fast
- Multiplication with fractions \(\geq 1\) \(\rightarrow\) intermediate numbers limited size
Questions – Which correctly computes \( \binom{n}{k} \) ?

Observation \[
\binom{n}{k} = \frac{n \cdot (n - 1) \cdot (n - 2) \cdots (n - k + 1)}{k \cdot (k - 1) \cdot (k - 2) \cdots 1}
\]

a) binomial_A

b) binomial_B

c) both

d) none

e) Don’t know

bionomial_iterative.py

```python
def binomial_A(n, k):
    result = 1
    for i in range(k):
        result = result * (n - i) // (k - i)
    return result

def binomial_B(n, k):
    result = 1
    for i in range(k)[::-1]:
        result = result * (n - i) // (k - i)
    return result
```
Recursively print all leaves of a tree

- Assume a recursively nested tuple represents a tree with strings as leaves

```python
def print_leaves(tree):
    if isinstance(tree, str):
        print("Leaf:", tree)
    else:
        for child in tree:
            print_leaves(child)

print_leaves(('a',('b','c')))  # Leaf: a
  | Leaf: b
  | Leaf: c
```
Question – How many times is `print_leaves` function called in the example?

```python
def print_leaves(tree):
    if isinstance(tree, str):
        print("Leaf:", tree)
    else:
        for child in tree:
            print_leaves(child)

print_leaves(('a',('b','c')))
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Leaf: a</td>
<td>Leaf: b</td>
<td>Leaf: c</td>
</tr>
</tbody>
</table>

- a) 3
- b) 4
- c) 5
- d) 6
- e) Don’t know
Collect all leaves of a tree in a set

```python
def collect_leaves_slow(tree):
    leaves = set()
    if isinstance(tree, str):
        leaves.add(tree)
    else:
        for child in tree:
            leaves |= collect_leaves_slow(child)
    return leaves

collect_leaves_slow(('a', ('b', 'c')))  # {'a', 'c', 'b'}
```
copies all labels from child from one set to another set
```python
def collect_leaves_wrong(tree, leaves=set()):
    if isinstance(tree, str):
        leaves.add(tree)
    else:
        for child in tree:
            collect_leaves_wrong(child, leaves)
    return leaves

def collect_leaves_right(tree, leaves=None):
    if leaves is None:
        leaves = set()
    if isinstance(tree, str):
        leaves.add(tree)
    else:
        for child in tree:
            collect_leaves_right(child, leaves)
    return leaves

print(collect_leaves_wrong(('a',('b','c'))))  # Output: {'a', 'c', 'b'}
print(collect_leaves_wrong(('d',('e','f'))))  # Output: {'b', 'e', 'a', 'f', 'c', 'd'}

print(collect_leaves_right(('a',('b','c'))))  # Output: {'b', 'a', 'c'}
print(collect_leaves_right(('d',('e','f'))))  # Output: {'f', 'd', 'e'}
```
```python
def collect_leaves(tree):
    leaves = set()

    def traverse(tree):
        nonlocal leaves  # can be omitted
        if isinstance(tree, str):
            leaves.add(tree)
        else:
            for child in tree:
                traverse(child)
    traverse(tree)
    return leaves

> collect_leaves(('a', ('b', 'c')))
| {'b', 'a', 'c'}
> collect_leaves(('d', ('e', 'f')))
| {'f', 'd', 'e'}
```
Maximum recursion depth?

- Python's maximum allowed recursion depth can be increased by

```python
import sys
sys.setrecursionlimit(1500)
```

```python
def f(x):
    print("#", x)
    f(x + 1)

f(1)
```

```
# 1
# 2
# 3
...
# 975
# 976
# 977
# 978
RecursionError: maximum recursion depth exceeded while pickling an object
```
Koch Curves

depth = 0

depth = 1

depth = 2

depth = 3

depth = 4
Koch Curves

import matplotlib.pyplot as plt
from math import sqrt

def koch(p, q, depth=3):
    if depth == 0:
        return [p, q]

    dx, dy = q[0] - p[0], q[1] - p[1]
    h = 1 / sqrt(12)
    p1 = p[0] + dx / 3, p[1] + dy / 3
    p2 = p[0] + dx / 2 - h * dy, p[1] + dy / 2 + h * dx
    p3 = p[0] + dx * 2 / 3, p[1] + dy * 2 / 3
    return (koch(p,  p1, depth - 1)[:-1] +
            koch(p1, p2, depth - 1)[:-1] +
            koch(p2, p3, depth - 1)[:-1] +
            koch(p3, q,  depth - 1))

points = koch((0, 1), (0, 0), depth=3)
X, Y = zip(*points)
plt.subplot(aspect='equal')
plt.plot(X, Y, 'r-')
plt.plot(X, Y, 'k.')
plt.show()
Recursion and iteration

- algorithm examples
## Standard 52-card deck

<table>
<thead>
<tr>
<th></th>
<th>Ace</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>Jack</th>
<th>Queen</th>
<th>King</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Clubs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>♣️</td>
<td>♣️</td>
<td>♣️</td>
<td>♣️</td>
<td>♣️</td>
<td>♣️</td>
<td>♣️</td>
<td>♣️</td>
<td>♣️</td>
<td>♣️</td>
<td>♣️</td>
<td>♣️</td>
<td>♣️</td>
</tr>
<tr>
<td><strong>Diamonds</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>♦️</td>
<td>♦️</td>
<td>♦️</td>
<td>♦️</td>
<td>♦️</td>
<td>♦️</td>
<td>♦️</td>
<td>♦️</td>
<td>♦️</td>
<td>♦️</td>
<td>♦️</td>
<td>♦️</td>
<td>♦️</td>
</tr>
<tr>
<td><strong>Hearts</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>♥️</td>
<td>♥️</td>
<td>♥️</td>
<td>♥️</td>
<td>♥️</td>
<td>♥️</td>
<td>♥️</td>
<td>♥️</td>
<td>♥️</td>
<td>♥️</td>
<td>♥️</td>
<td>♥️</td>
<td>♥️</td>
</tr>
<tr>
<td><strong>Spades</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>♠️</td>
<td>♠️</td>
<td>♠️</td>
<td>♠️</td>
<td>♠️</td>
<td>♠️</td>
<td>♠️</td>
<td>♠️</td>
<td>♠️</td>
<td>♠️</td>
<td>♠️</td>
<td>♠️</td>
<td>♠️</td>
</tr>
</tbody>
</table>

Selection sort

```
selection_sort.py

def selection_sort(L):
    unsorted = L[:]
    result = []

    while unsorted:
        e = min(unsorted)
        unsorted.remove(e)
        result.append(e)

    return result
```

- **min** and `.remove` scan the remaining `unsorted` list for each element moved to `result`
- Order $|L|^2$ comparisons
Sorting a pile of cards (Merge sort)

- If one card in pile, i.e. pile is sorted
- Otherwise
  1) **Split** pile into two piles, **left** and **right**, of approximately same size
  2) Sort **left** and **right** recursively (independently)
  3) **Merge** **left** and **right** (which are sorted)
merge_sort.py

def merge_sort(L):
    n = len(L)
    if n <= 1:
        return L[:]
    else:
        mid = n // 2
        left, right = L[:mid], L[mid:]
        return merge(merge_sort(left), merge_sort(right))

def merge(A, B):
    n = len(A) + len(B)
    C = n * [None]
    a, b = 0, 0
    for c in range(n):
        if a < len(A) and (b == len(B) or A[a] < B[b]):
            C[c] = A[a]
            a = a + 1
        else:
            C[c] = B[b]
            b = b + 1
    return C
Question – Depth of recursion for 52 elements

a) 1
b) 2
c) 3
d) 4
e) 5
f) 6
g) 7
h) 8
i) 9
j) 10
k) Don’t know

Depth 4 for 8 elements
Question – Order of comparisons by Merge sort?

a) \( \sim n \)
b) \( \sim n\sqrt{n} \)
c) \( \sim n \log_2 n \)
d) \( \sim n^2 \)
e) \( \sim n^3 \)
f) Don’t know

```
merge_sort.py

def merge_sort(L):
    n = len(L)
    if n <= 1:
        return L[:]
    else:
        mid = n // 2
        left, right = L[:mid], L[mid:]
        return merge(merge_sort(left), merge_sort(right))

def merge(A, B):
    n = len(A) + len(B)
    C = n * [None]
    a, b = 0, 0
    for c in range(n):
        if a < len(A) and (b == len(B) or A[a] < B[b]):
            C[c] = A[a]
            a = a + 1
        else:
            C[c] = B[b]
            b = b + 1
    return C
```
Merge sort without recursion

- Start with piles of size one
- Repeatedly merge two smallest piles

```python
merge_sort.py

def merge_sort_iterative(L):
    Q = [[x] for x in L]
    while len(Q) > 1:
        Q.insert(0, merge(Q.pop(), Q.pop()))
    return Q[0]

from collections import deque
def merge_sort_deque(L):
    Q = deque([[x] for x in L])
    while len(Q) > 1:
        Q.appendleft(merge(Q.pop(), Q.pop()))
    return Q[0]
```

merge_sort_iterative([7, 1, 9, 3, -2, 5])

Values of Q in while-loop
[[7], [1], [9], [3], [-2], [5]]
[[2, 5], [7], [1], [9], [3]]
[[3, 9], [-2, 5], [7], [1]]
[[1, 7], [3, 9], [-2, 5]]
[[-2, 3, 5, 9], [1, 7]]
[[-2, 1, 3, 5, 7, 9]]

Note: Lists in Q appear in non-increasing length order, where longest ≤ 2·shortest
Question – Number of iterations of while-loop?

merge_sort_iterative([7, 1, 9, 3, -2, 5])

- a) 1
- b) 2
- c) 3
- d) 4
- e) 5
- f) 6
- g) 7
- h) Don’t know

merge_sort.py

```python
def merge_sort_iterative(L):
    Q = [[x] for x in L]
    while len(Q) > 1:
        Q.insert(0, merge(Q.pop(), Q.pop()))
    return Q[0]
```
Quicksort (randomized)

```python
import random

def quicksort(L):
    if len(L) <= 1:
        return L[:]
    idx = random.randint(0, len(L)-1)
    pivot = L[idx]
    other = L[:idx] + L[idx+1:]

    small = [e for e in other if e < pivot]
    large = [e for e in other if e >= pivot]

    return quicksort(small) + [pivot] + quicksort(large)
```

orders |$|L|\cdot \log_2 |L|$ comparisons, expected
## Sorting comparison (single run)

| $|L|$ | Selection sort | Merge sort Recursive | Merge sort Iterative | Merge sort Deque | Quicksort |
|---|---|---|---|---|---|
| $2^{10}$ | 0.02 | 0.00 | 0.01 | 0.00 | 0.00 |
| $2^{11}$ | 0.08 | 0.01 | 0.02 | 0.02 | 0.01 |
| $2^{12}$ | 0.29 | 0.03 | 0.05 | 0.04 | 0.02 |
| $2^{13}$ | 1.17 | 0.07 | 0.13 | 0.06 | 0.04 |
| $2^{14}$ | 4.62 | 0.14 | 0.28 | 0.14 | 0.08 |
| $2^{15}$ | 18.78 | 0.29 | 0.54 | 0.28 | 0.20 |
| $2^{16}$ | 74.27 \* 4 | 0.64 | 1.92 | 0.89 | 0.33 |
| $2^{17}$ | 1.48 | 5.74 | 1.62 | 0.69 |
| $2^{18}$ | 3.11 | 20.85 \* 4 | 3.66 | 1.49 |
| $2^{19}$ | 6.41 | 79.05 \* 4 | 7.91 | 3.52 |
| $2^{20}$ | 13.52 | 15.08 | 17.38 \* 2 | 7.83 |
| $2^{21}$ | 28.30 \* 2 | 31.32 \* 2 | 63.52 \* 2 | 40.80 \* 2 |
| $2^{22}$ | 59.60 \* 2 |
Sorting comparison

![Sorting algorithms graph]
Find zero

- Given a list L of integers starting with a negative and ending with a positive integer, and where \(|L[i+1] - L[i]| \leq 1\), find the position of a zero in L.

L = [-5, -4, -3, -3, -4, -3, -2, -1, 0, 1, 2, 1, 0, -1, -2, -1, 0, 1, 2, 3, 2]
```python
def find_zero_loop(L):
    i = 0
    while L[i] != 0:
        i += 1
    return i

def find_zero_enumerate(L):
    for idx, e in enumerate(L):
        if e == 0:
            return idx

def find_zero_index(L):
    return L.index(0)

def find_zero_binary_search(L):
    low = 0
    high = len(L) - 1
    while True:
        # L[low] < 0 < L[high]
        mid = (low + high) // 2
        if L[mid] == 0:
            return mid
        elif L[mid] < 0:
            low = mid
        else:
            high = mid

def find_zero_recursive(L):
    def search(low, high):
        mid = (low + high) // 2
        if L[mid] == 0:
            return mid
        elif L[mid] < 0:
            return search(mid, high)
        else:
            return search(low, mid)
    return search(0, len(L) - 1)
```

| Function (|L| = 10^6)         | Time, sec |
|---------------|-----------|
| find_zero_loop | 0.13      |
| find_zero_enumerate | 0.10     |
| find_zero_index    | 0.015    |
| find_zero_binary_search | 0.000015 |
| find_zero_recursive | 0.000088 |

```
```
Greatest Common Divisor (GCD)

**Notation**
$x^\uparrow y$ denotes $y$ is divisible by $x$, e.g. $3^\uparrow 12$ i.e. $y = a \cdot x$ for some integer $a$.

**Definition**
$\text{gcd}(m, n) = \max \{ x \mid x^\uparrow m \text{ and } x^\uparrow n \}$

**Fact**
If $x^\uparrow y$ and $x^\uparrow z$ then $x^\uparrow (y+z)$ and $x^\uparrow (y-z)$

**Observation**
(Recursive definition)
$\text{gcd}(m, n) = \begin{cases} m & \text{if } m = n \\ \text{gcd}(m, n - m) & \text{if } m < n \\ \text{gcd}(m - n, n) & \text{if } m > n \end{cases}$

<table>
<thead>
<tr>
<th>m</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>24</td>
</tr>
<tr>
<td>66</td>
<td>24</td>
</tr>
<tr>
<td>42</td>
<td>24</td>
</tr>
<tr>
<td>18</td>
<td>24</td>
</tr>
<tr>
<td>18</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

$\text{gcd}(90, 24)$
# Greatest Common Divisor (GCD)

<table>
<thead>
<tr>
<th>File</th>
<th>Code</th>
</tr>
</thead>
</table>
| **gcd.py**               | def gcd(m, n):
  while n != 0:
    if n > m:
      n = n - m
    else:
      m = m - n
  return m                   |
| **gcd_recursive.py**     | def gcd(m, n):
  if n == 0:
    return m
  else:
    return gcd(n, m % n)      |
| **gcd_recursive_one_line.py** | def gcd(m, n):
  return m if n == 0 else gcd(n, m % n) |
| **gcd_slow.py**          | def gcd(m, n):
  while m != n:
    if n > m:
      n = n - m
    else:
      m = m - n
  return m                   |
| **gcd_slow_recursive.py**| def gcd(m, n):
  if m == n:
    return m
  elif m > n:
    return gcd(m - n, n)
  else:
    return gcd(m, n - m)      |
Permutations

- Generate all permutations of a list \(L\) as tuples

**Python shell**

```python
\$ permutations(['a','b','c'])
[('a', 'b', 'c'), ('b', 'a', 'c'), ('b', 'c', 'a'), ('a', 'c', 'b'), ('c', 'a', 'b'), ('c', 'b', 'a')]
```

**permutations.py**

```python
def permutations(L):
    if len(L) == 0:
        return [L[:]]
    else:
        P = permutations(L[1:])
        return [p[:i] + L[:1] + p[i:] for p in P for i in range(len(L))]
```

- An implementation of "permutations" exists in the "itertools" library
## Maze solver

### Input
- First line `#rows and #columns`
- Following `#rows` lines contain strings containing `#column` characters
- There are exactly one 'A' and one 'B'
- '.' are free cells and '#' are blocked cells

### Output
- Print whether there is a path from 'A' to 'B' or not

---

**maze input**

<table>
<thead>
<tr>
<th>11</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>#######</td>
<td>A#######</td>
</tr>
</tbody>
</table>
maze_solver.py

```python
def explore(i, j):
    global solution, visited
    if (0 <= i < n and 0 <= j < m and maze[i][j] != '#' and not visited[i][j]):
        visited[i][j] = True
        if maze[i][j] == 'B':
            solution = True
        explore(i - 1, j)
        explore(i + 1, j)
        explore(i, j - 1)
        explore(i, j + 1)

def find(symbol):
    for i in range(n):
        j = maze[i].find(symbol)
        if j >= 0:
            return (i, j)

n, m = [int(x) for x in input().split()]
maze = [input() for i in range(n)]
solution = False
visited = [m*[False] for i in range(n)]
explore(*find('A'))
if solution:
    print("path from A to B exists")
else:
    print("no path")
```
Maze solver (iterative)

```python
def explore(i, j):
    global solution, visited
    Q = [(i, j)] # cells to visit
    while Q:
        i, j = Q.pop()
        if (0 <= i < n and 0 <= j < m and
            maze[i][j] != '#' and not visited[i][j]):
            visited[i][j] = True
            if maze[i][j] == 'B':
                solution = True
            Q.append((i - 1, j))
            Q.append((i + 1, j))
            Q.append((i, j - 1))
            Q.append((i, j + 1))

def find(symbol):
    for i in range(n):
        j = maze[i].find(symbol)
        if j >= 0:
            return (i, j)

n, m = [int(x) for x in input().split()]
maze = [input() for i in range(n)]
solution = False
visited = [m*[False] for i in range(n)]
explore(*find('A'))
if solution:
    print("path from A to B exists")
else:
    print("no path")
```
Question – How difficult is the triplet project on a scale 1 – 10?

a) 1 (I’m offended by how trivial the project was)
b) 2 (very easy)
c) 3 (a quite standard review exercise)
d) 4 (not too complicated, got some known concepts repeated)
e) 5 (good exercise to repeat standard programming techniques)
f) 6 (had to use more advanced techniques in a familiar way)
g) 7 (quite complicated, but manageable)
h) 8 (very abstract exercise, using complicated language constructs)
i) 9 (very complicated – barely manageable spending all my time)
j) 10 (this is a research project – could be an MSc thesis/PhD project)
k) 25 (this is wayyy too complicated for a university course)
Functions as objects

- lambda
- higher-order functions
- map, filter, reduce
Aliasing functions – both user defined and built-in

Python shell

```python
> def square(x):
    return x * x
> square
  | <function square at 0x0329A390>
> square(8)
  | 64
> kvadrat = square
> kvadrat(5)
  | 25
> len
  | <built-in function len>
> length = len
> length([1, 2, 3])
  | 3
```
Functions as values

**square_or_double.py**

```python
def square(x):
    return x * x

def double(x):
    return 2 * x

while True:
    answer = input("square or double ? ")
    if answer == "square":
        f = square
        break
    if answer == "double":
        f = double
        break

answer = input("numbers: ")
L_in = [int(x) for x in answer.split()]
L_out = [f(x) for x in L_in]
print(L_out)
```

**Python shell**

```
square or double ? square
numbers: 3 6 7 9
[9, 36, 49, 81]
square or double ? double
numbers: 2 3 4 7 9
[4, 6, 8, 14, 18]
```

- `f` will refer to one of the functions *square* or *double*
- *square* and *double* refer to call the function `f` is referring to with argument `x`
Functions as values and namespaces

- `what_says` is a function returning a function (`say`)
- Each call to `what_says` with a single string as its argument creates a new `say` function with the current `name` argument in its namespace
- In each call to a an instance of a `say` function, `name` refers to the string in the namespace when the function was created, and `message` is the string given as an argument in the call

**say.py**

```python
def what_says(name):
    def say(message):
        print(name, "says:", message)
    return say

alice = what_says("Alice")
peter = what_says("Peter")

alice("Where is Peter?")
peter("I am here")
```

**Python shell**

<table>
<thead>
<tr>
<th>Alice says: Where is Peter?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peter says: I am here</td>
</tr>
</tbody>
</table>
def f(x):
    def g(y):
        nonlocal x
        x = x + 1
        return x + y
    return g
a = f(3)
b = f(6)
print([a(3), b(2), a(4)])

a) [7, 7, 10]
b) [7, 9, 8]
c) [7, 9, 9]
d) [7, 9, 12]
e) [7, 10, 10]
f) Don’t know
map

- `map(function, list)` applies the function to each element of the sequence `list`.
- `map(function, list_1, ..., list_k)` requires function to take `k` arguments, and creates a sequence with the `i`'th element being `function(list_1[i], ..., list_k[i])`.

Python shell

```python
> def square(x):
    return x*x
> list(map(square, [1,2,3,4,5]))
[1, 4, 9, 16, 25]
> def triple_sum(x, y, z):
    return x + y + z
> list(map(triple_sum, [1,2,3], [4,5,6], [7,8,9]))
[12, 15, 18]
> list(map(triple_sum, *zip(*[[(1,4,7), (2,5,8), (3,6,9)]])))
[12, 15, 18]
```
A list \( L \) can be sorted using `sorted(L)`.

A user defined order on the elements can be defined by providing a function using the keyword argument `key`, that maps elements to values with some default ordering.

```python
# Python shell

def length_square(p):
    x, y = p
    return x**2 + y**2  # no sqrt

L = [(5, 3), (2, 5), (1, 9), (2, 2), (3, 4)]
list(map(length_square, L))  # [34, 29, 82, 8, 25]
sorted(L)  # default lexicographical order
[(1, 9), (2, 2), (2, 5), (3, 4), (5, 3)]
sorted(L, key=length_square)  # order by length
[(2, 2), (3, 4), (2, 5), (5, 3), (1, 9)]
```

[https://docs.python.org/3/library/functions.html#sorted](https://docs.python.org/3/library/functions.html#sorted)
Question – What list does `sorted` produce?

```python
sorted([2, 3, -1, 5, -4, 0, 8, -6], key=abs)
```

- a) `[-6, -4, -1, 0, 2, 3, 5, 8]`
- b) `[0, 2, 3, 5, 8, -1, -4, -6]`
- c) `[0, -1, 2, 3, -4, 5, -6, 8]`
- d) `[8, 5, 3, 2, 0, -1, -4, -6]`
- e) `[0, 1, 2, 3, 4, 5, 6, 8]`
- f) Don’t know

Python shell:
```
> abs(7)
7
> abs(-42)
42
```
filter

- `filter(function, list)` returns the subsequence of `list` where `function` evaluates to true

- Essentially the same as

  \[ [x \text{ for } x \text{ in } list \text{ if } function(x)] \]

**Python shell**

```
> def odd(x):
    return x % 2 == 1

> filter(odd, range(10))
| <filter object at 0x03970FD0>
> list(filter(odd, range(10)))
| [1, 3, 5, 7, 9]
```
reduce (in module functools)

- Python’s “reduce” function is in other languages often denoted “foldl”

\[
\text{reduce}(f, [x_1, x_2, x_3, \ldots, x_k]) = f(\ldots f(f(x_1, x_2), x_3)\ldots, x_k)
\]

**Python shell**

```
> from functools import reduce
> def power(x, y):
>     return x ** y
> reduce(power, [2, 2, 2, 2, 2])
| 65536
```
lambda (anonymous functions)

- If you need to define a short function, that returns a value, and the function is only used once in your program, then a lambda function might be appropriate:

  lambda arguments: expression

- Creates a function with no name that takes zero or more arguments, and returns the value of the single expression

```python
Python shell
> f = lambda x, y : x + y
> f(2, 3)
| 5
> list(filter(lambda x: x % 2, range(10)))
| [1, 3, 5, 7, 9]
```
Examples: sorted using lambda

```python
L = ['AHA', 'Oasis', 'ABBA', 'Beatles', 'AC/DC', 'B. B. King', 'Bangles', 'Alan Parsons']

# Sort by length, secondary after input position (default, known as stable)
sorted(L, key=len)
['AHA', 'ABBA', 'Oasis', 'AC/DC', 'Beatles', 'Bangles', 'B. B. King', 'Alan Parsons']

# Sort by length, secondary alphabetically
sorted(L, key=lambda s: (len(s), s))
['AHA', 'ABBA', 'AC/DC', 'Oasis', 'Bangles', 'Beatles', 'B. B. King', 'Alan Parsons']

# Sort by most 'a's, if equal by number of 'b's, etc.
sorted(L, key=lambda s: sorted([a.lower() for a in s if a.isalpha()]) + ['~'])
['Alan Parsons', 'ABBA', 'AHA', 'Beatles', 'Bangles', 'AC/DC', 'Oasis', 'B. B. King']

sorted([a.lower() for a in 'Beatles' if a.isalpha()]) + ['~'])
['a', 'b', 'e', 'e', 'l', 's', 't', '~']
```
min and max

- Similarly to `sorted`, the functions `min` and `max` take a keyword argument `key`, to map elements to values with some default ordering.

```python
Python shell
> max(['w', 'xyz', 'abcd', 'uv'])
 | 'xyz'
> max(['w', 'xyz', 'abcd', 'uv'], key=len)
 | 'abcd'
> sorted([210, 13, 1010, 30, 27, 103], key=lambda x: str(x)[::-1])
 | [1010, 210, 30, 103, 13, 27]
> min([210, 13, 1010, 30, 27, 103], key=lambda x: str(x)[::-1])
 | 1010
```

docs.python.org/3/library/functions.html#min
History of lambda in programming languages

- Lambda calculus invented by Alonzo Church in 1930s
- Lisp has had lambdas since 1958
- C++ got lambdas in C++11 in 2011
- Java first got lambdas in Java 8 in 2014
- Python has had lambdas since Version 1.0 in 1994
def linear_function(a, b):
    return lambda x: a * x + b

def degree_two_polynomial(a, b, c):
    def evaluate(x):
        return a * x**2 + b * x + c
    return evaluate

def polynomial(coefficients):
    return lambda x: sum([c * x**p for p, c in enumerate(coefficients)])

def combine(f, g):
    def evaluate(*args, **kwargs):
        return f(g(*args, **kwargs))
    return evaluate

f = linear_function(2, 3)
for x in [0, 1, 2]:
    print("f(\(\%d\)) = \(\%s\) \% (\(\%d\), f(\(\%d\)))" % (x, f(x)))

p = degree_two_polynomial(1, 2, 3)
for x in [0, 1, 2]:
    print("p(\(\%d\)) = \(\%s\) \% (\(\%d\), p(\(\%d\)))" % (x, p(x)))

print("polynomial([3, 2, 1])(2) =", polynomial([3, 2, 1])(2))

h = combine(abs, lambda x, y: x - y)
print("h(3, 5) =", h(3, 5))

| f(0) = 3 |
| f(1) = 5 |
| f(2) = 7 |
| p(0) = 3 |
| p(1) = 6 |
| p(2) = 11 |
| polynomial([3, 2, 1])(2) = 11 |
| h(3, 5) = 2 |
Question – What value is $h(1)$?

```python
# linear_combine.py

def combine(f, g):
    def evaluate(*args, **kwargs):
        return f(g(*args, **kwargs))
    return evaluate

def linear_function(a, b):
    return lambda x: a * x + b

f = linear_function(2, 3)
g = linear_function(4, 5)

h = combine(f, g)

print(h(1))
```

a) 5  

b) 9  

c) 16  

d) 21  

e) 25  

f) Don’t know
Namespace example

```python
def combine(f, g):
    def evaluate(*args, **kwargs):
        return f(g(*args, **kwargs))
    return evaluate

def linear_function(a, b):
    return lambda x: a * x + b

f = linear_function(2, 3)
g = linear_function(4, 5)
h = combine(f, g)
print(h(1))
```

The function `partial` from the module `functools` fixes/binds/freezes a subset of the parameters of a function

Roughly equivalent to the definition in the example

**partial.py**
```
def partial(f, *args, **kwargs):
    return lambda *a, **kw: f(*args, *a, **kwargs, **kw)

def f(x, y, z):
    return x + 2 * y + 3 * z

g = partial(f, 7)
h = partial(f, 2, 1)
k = partial(g, 1, 2)

print(g(2, 1))  # 7 + 2 * 2 + 3 * 1 = 14
print(h(3))  # 2 + 2 * 1 + 3 * 3 = 13
print(k())  # 7 + 2 * 1 + 3 * 2 = 15
```
partial (trace of computation)

```python
--- partial_trace.py ---
def partial(fn, *args):
    def new_f(*a):
        print(f'new_f: fn={fn.__name__}, args={args}, a={a}"
        answer = fn(*args, *a)
        print(f'answer={answer}"
        return answer
    return new_f

def f(x, y, z):
    print(f'f({x},{y},{z})"
    return x + 2 * y + 3 * z

g = partial(f, 7)
h = partial(f, 2, 1)
k = partial(g, 1, 2)

print(f'g(2, 1)=\n'    # 7 + 2 * 2 + 3 * 1 = 14
print(f'h(3)=\n'    # 2 + 2 * 1 + 3 * 3 = 13
print(f'k()=\n'    # 7 + 2 * 1 + 3 * 2 = 15
```
Object oriented programming

- classes, objects
- self
- construction
- encapsulation
Object Oriented Programming

- **Programming paradigm**, other paradigms are e.g.
  - *functional programming* where the focus is on functions, lambda’s and higher order functions, and
  - *imperative programming* focusing on sequences of statements changing the state of the program

- Core concepts are **objects**, **methods** and **classes**,
  - allowing one to construct *abstract data types*, i.e. user *defined types*
  - objects have states
  - methods manipulate objects, defining the interface of the object to the rest of the program

- OO supported by **many programming languages**, including Python
Object Oriented Programming - History
(selected programming languages)

Mid 1960’s  **Simular 67**
(Ole-Johan Dahl and Kristen Nygaard, Norsk Regnesentral Oslo)
Introduced classes, objects, virtual procedures

1970’s  **Smalltalk** (Alan Kay, Dan Ingalls, Adele Goldberg, Xerox PARC)
Object-oriented programming, fully dynamic system
(opposed to the static nature of Simula 67)

1985  **Eiffel** (Bertrand Meyer, Eiffel Software)
Focus on software quality, capturing the full software cycle

1985  **C++** (Bjarne Stroustrup [MSc Aarhus 1975], AT&T Bell Labs)

1995  **Java** (James Gosling, Sun)

2000  **C#** (Anders Hejlsberg (studied at DTU) et al., Microsoft)

1991  **Python** (Guido van Rossum)
Multi-paradigm programming language, fully dynamic system

**Note:** Java, C++, Python, C# are among Top 5 on TIOBE January 2020 index of popular languages (only non OO language among Top 5 was C)
Design Patterns (not part of this course)

reoccurring patterns in software design

The Classic book 1994
(C++ cookbook)

A very alternative book 2004
(Java, very visual)

Java cookbook 2003
Java textbook 2004
Java textbook 2010

...and many more books on the topic of Design Patterns, also with Python
Some known classes, objects, and methods

<table>
<thead>
<tr>
<th>Type / class</th>
<th>Objects</th>
<th>Methods (examples)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>0 -7 42 1234567</td>
<td>__add__(x), __eq__(x), __str__(x)</td>
</tr>
<tr>
<td>str</td>
<td>&quot;&quot; 'abc' '12_a'</td>
<td>isdigit(), .lower(), __len__(x)</td>
</tr>
<tr>
<td>list</td>
<td>[] [1,2,3] ['a', 'b', 'c']</td>
<td>append(x), .clear(), __mul__(x)</td>
</tr>
<tr>
<td>dict</td>
<td>{'foo': 42, 'bar': 5}</td>
<td>.keys(), .get(), __getitem__(x)</td>
</tr>
<tr>
<td>NoneType</td>
<td>None</td>
<td>__str__(x)</td>
</tr>
</tbody>
</table>

**Python shell**

```
> 5 + 7  # + calls \_\_add\_\_(7) | 12
> (5).\_\_add\_\_(7)  # eq. to 5 + 7 | 12
> (7).\_\_eq\_\_(7)  # eq. to 7 == 7 | True
> 'aBCd'.lower() | 'abcd'
> 'abcde'.\_\_len\_\_(x) # \_\_len\_\_(x) called by len(...) | 5
> ['x', 'y'].\_\_mul\_\_(2) | ['x', 'y', 'x', 'y']
> {'foo': 42}.\_\_getitem\_\_('foo') # eq. to {'foo': 42}['foo'] | 42
> None.\_\_str\_\_(x) # used by str(...) | 'None'
```

Example:
The function `str(obj)` calls the methods `obj.\_\_str\_\_(x)` or `obj.\_\_repr\_\_(x)`, if `obj.\_\_str\_\_` does not exist.
Classes and Objects

class Student
set_name(name)
set_id(student_id)
get_name()
get_id()

Creating instances of class Student using constructor
Student()

objects
(instances)

student_SM
name = 'Scrooge McDuck'
id = '777'

student_MM
name = 'Mickey Mouse'
id = '243'

student_DD
name = 'Donald Duck'
id = '107'

data attributes

docs.python.org/3/tutorial/classes.html
Using the Student class

```python
student_DD = Student()
student_MM = Student()
student_SM = Student()

student_DD.set_name('Donald Duck')
student_DD.set_id('107')

student_MM.set_name('Mickey Mouse')
student_MM.set_id('243')

student_SM.set_name('Scrooge McDuck')
student_SM.set_id('777')

students = [student_DD, student_MM, student_SM]

for student in students:
    print(student.get_name(),
          "has id",
          student.get_id())
```

Python shell

| Donald Duck has id 107 |
| Mickey Mouse has id 243 |
| Scrooge McDuck has id 777 |

- Call **constructor** for class **Student**. Each call returns a new **Student** object.
- Call class methods to set data attributes
- Call class methods to read data attributes
```python
from student import Student

class Student:
    def set_name(self, name):
        self.name = name
    def set_id(self, student_id):
        self.id = student_id
    def get_name(self):
        return self.name
    def get_id(self):
        return self.id
```

**Note** In other OO programming languages, the explicit reference to `self` is not required (in Java and C++ `self` is the keyword `this`).
When are object attributes initialized?

Python shell

```python
> x = Student()
> x.set_name("Gladstone Gander")
> x.get_name()
'Gladstone Gander'
> x.get_id()
AttributeError: 'Student' object has no attribute 'id'
```

- Default behaviour of a class is that instances are created with no attributes defined, but has access to the attributes / methods of the class.

- In the previous class Student both the name and id attributes were first created when set by set_name and set_id, respectively.
Class construction and __init__

- When an object is created using `class_name()` it’s initializer method __init__ is called.
- To initialize objects to contain default values, (re)define this function.

```python
student.py

class Student:
    def __init__(self):
        self.name = None
        self.id = None

... previous method definitions ...
```
Question – What is printed?

Python shell

```python
> class C:
>     def __init__(self):
>         self.v = 0
>     def f(self):
>         self.v = self.v + 1
>         return self.v
> x = C()
> print(x.f() + x.f())
```

a) 1
b) 2

😊
c) 3
d) 4
e) 5
f) Don’t know
__init__ with arguments

- When creating objects using `class_name(args)` the initializer method is called as `__init__(args)`
- To initialize objects to contain default values, (re)define this function to do the appropriate initialization

```python
student.py
class Student:
    def __init__(self, name=None, student_id=None):
        self.name = name
        self.id = student_id

... previous method definitions ...
```

```
Python shell
> p = Student("Pluto")
> print(p.get_name())
| Pluto
> print(p.get_id())
| None
```
Are accessor and mutator methods necessary?

No - but good programming style

Python shell

```python
> p = Pair(3, 5)
> p.sum()
| 8
> p.set_a(4)
> p.sum()
| 9
> p.a  # access object attribute
| 4
> p.b = 0  # update object attribute
| 0  # the _sum not updated
```

pair.py

```python
class Pair:
    """ invariant: the_sum = a + b """
    def __init__(self, a, b):
        self.a = a
        self.b = b
        self.the_sum = self.a + self.b
    def set_a(self, a):
        self.a = a
        self.the_sum = self.a + self.b
    def set_b(self, b):
        self.b = b
        self.the_sum = self.a + self.b
    def sum(self):
        return self.the_sum
```
Defining order on instances of a class (sorting)

- To define an order on objects, define the "<" operator by defining `__lt__`

- When "<" is defined a list \( L \) of students can be sorted using `sorted(L)` and `L.sort()`

```
student.py

class Student:
    def __lt__(self, other):
        return self.id < other.id

... previous method definitions ...

Python shell

> student_DD < student_MM
| True
> [x.id for x in students]
| ['243', '107', '777']
> [x.id for x in sorted(students)]
| ['107', '243', '777']
```
Converting objects to \texttt{str}

- To be able to convert an object to a string using \texttt{str(object)}, define the method \texttt{__str__}

- \texttt{__str__} is e.g. used by \texttt{print}

```
student.py

class Student:
    def __str__(self):
        return "Student['%s', '%s']" % (self.name, self.id)

... previous method definitions ...
```

```
Python shell

> print(student_DD)  # without \_\_str\_\_
| <__main__.Student object at 0x03AB6B90>
> print(student_DD)  # with \_\_str\_\_
| Student['Donald Duck', '107']
```
Nothing is private in Python

- Python does not support hiding information inside objects.
- Recommendation is to start attributes with underscore, if these should be used only locally inside a class, i.e. be considered ”private”.
- PEP8: “Use one leading underscore only for non-public methods and instance variables.”

```python
class My_Class:
    def set_xy(self, a, b):
        self._x = a
        self._y = b

    def get_sum(self):
        return self._x + self._y

obj = My_Class()
obj.set_xy(3, 5)
print("Sum =", obj.get_sum())
print("_x =", obj._x)
```

Python shell

```
| Sum = 8 |
| _x = 3  |
```
C++ private, public

C++ vs Python
1. argument types
2. return types
3. void = NoneType
4. private/public access specifier
5. types of data attributes
6. data attributes must be defined in class
7. object creation
8. no self in class methods

```cpp
#include <iostream>
using namespace std;

class My_Class {
private:
  int x, y;
public:
  void set_xy(int a, int b) {
    x = a;
    y = b
  }
  int get_sum() {
    return x + y;
  }
};

int main() {
  My_Class obj;
  obj.set_xy(3, 5);
  cout << "Sum = " << obj.get_sum() << endl;
  cout << "x = " << obj.x << endl;
}
```

expected output:
```
Sum = 8
x = 3
```
Java vs Python

1. argument types
2. return types
3. void = NoneType
4. private/public access specifier
5. types of data attributes
6. data attributes must be defined in class
7. object creation
8. no self in class methods
Name mangling (partial privacy)

- Python handles references to class attributes inside a class definition with *at least two leading underscores and at most one trailing underscore* in a special way: `__attribute` is textually replaced by `_classname__attribute`.

- Note that [Guttag, p. 126] states "that attribute is not visible outside the class" – which only is partially correct (see example).

```
name_mangeling.py

class MySecretBox:
    def __init__(self, secret):
        self.__secret = secret

Python shell

> x = MySecretBox(42)
> print(x.__secret)
| AttributeError: 'MySecretBox' object has no attribute '__secret'
> print(x._MySecretBox__secret)
| 42
```
Class attributes

<table>
<thead>
<tr>
<th>Class</th>
<th>Student</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>next_id</strong></td>
<td>3</td>
</tr>
<tr>
<td>set_name(name)</td>
<td></td>
</tr>
<tr>
<td>set_id(student_id)</td>
<td></td>
</tr>
<tr>
<td>get_name()</td>
<td></td>
</tr>
<tr>
<td>get_id()</td>
<td></td>
</tr>
</tbody>
</table>

- `obj.attribute` first searches the objects attributes to find a match, if no match, continuous to search the attributes of the class
- Assignments to `obj.attribute` are always to the objects attribute (possibly creating the attribute)
- Class attributes can be accessed directly as `class.attribute` (or `obj.__class__.attribute`)
Class data attribute

- **next_id** is a class attribute
- Accessed using `Student.next_id`
- The lookup can be replaced with `self.next_id`, since only the class has this attribute, looking up in the object will be propagated to a lookup in the class attributes
- In the update it is crucial that we update the class attribute, since otherwise the incremented value will be assigned as an object attribute
  (What will the result be?)

```python
class Student:
    next_id = 1  # class attribute
    def __init__(self, name):
        self.name = name
        self.id = str(Student.next_id)  # ① Student.next_id += 1
    def get_name(self):
        return self.name
    def get_id(self):
        return self.id

students = [Student('Scrooge McDuck'),
            Student('Donald Duck'),
            Student('Mickey Mouse')]
for student in students:
    print(student.get_name(),
          "has student id",
          student.get_id())
```

**Python shell**

- Scrooge McDuck has student id 1
- Donald Duck has student id 2
- Mickey Mouse has student id 3
Question – What does `obj.get()` return?

Python shell

```python
> class MyClass:
    x = 2

    def get(self):
        self.x = self.x + 1
        return MyClass.x + self.x

> obj = MyClass()
> print(obj.get())
| ?
```

- a) 4
- b) 5
- c) 6
- d) UnboundLocalError
- e) Don’t know
Class data attribute example (in Python)

- Note that `My_Class.x` and `self.x` refer to the same class attribute (since `self.x` has never been assigned a value)
Java static

- In Java class attributes, i.e. attribute values shared by all instances, are labeled static.
- Python allows both class and instance attributes with the same name – in Java at most one of them can exist.

```java
static_attributes.java

class My_Class {
    public static int x = 1;
    public void inc() { x += 1; }
}

class static_attributes {
    public static void main(String args[]){
        My_Class obj1 = new My_Class();
        My_Class obj2 = new My_Class();
        obj1.inc();
        obj2.inc();
        System.out.println(obj1.x);
        System.out.println(obj2.x);
    }
}
```

Java output

| 3 |
| 3 |
C++ static

- In C++ class attributes, i.e. attribute values shared by all instances, are labeled static
- ISO C++ forbids in-class initialization of non-const static member
- Python allows both class and instance attributes with the same name – in C++ at most one of them can exist

```cpp
#include <iostream>
using namespace std;

class My_Class {
public:
    static int x; // "= 1" is not allowed
    void inc() { x += 1; };
};

int My_Class::x = 1; // class initialization

int main(){
    My_Class obj1;
    My_Class obj2;
    obj1.inc();
    obj2.inc();
    cout << obj1.x << endl;
    cout << obj2.x << endl;
}
```

C++ output

| 3
| 3
A simple usage of class data attributes is to store a set of constants (but there is nothing preventing anyone to change these values)

```python
Python shell
> class Color:
    RED = "ff0000"
    GREEN = "00ff00"
    BLUE = "0000ff"
> Color.RED
| 'ff0000'
```
PEP8 Style Guide for Python Code (some quotes)

- **Class names** should normally use the *CapWords* convention.
- Always use `self` for the first argument to instance methods.
- Use one *leading underscore* only for non-public methods and instance variables.
- For **simple public data attributes**, it is best to expose just the attribute name, *without complicated accessor/mutator methods*.
- Always decide whether a class's methods and instance variables (collectively: "attributes") should be **public** or **non-public**. If in doubt, choose non-public; it's easier to make it public later than to make a public attribute non-public.

[www.python.org/dev/peps/pep-0008/](http://www.python.org/dev/peps/pep-0008/)
Some methods many class have

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__eq__</code> (self, other)</td>
<td>Used to test if two elements are equal</td>
</tr>
<tr>
<td></td>
<td>Two elements where <code>__eq__</code> is true must have equal <code>__hash__</code></td>
</tr>
<tr>
<td><code>__str__</code> (self)</td>
<td>Used by <code>print</code></td>
</tr>
<tr>
<td><code>__repr__</code> (self)</td>
<td>Used e.g. for printing to shell</td>
</tr>
<tr>
<td></td>
<td>(usually something that is a valid Python expression for <code>eval()</code>)</td>
</tr>
<tr>
<td><code>__len__</code> (self)</td>
<td>Length (integer) of object, e.g. lists, strings, tuples, sets, dictionaries</td>
</tr>
<tr>
<td><code>__doc__</code> (self)</td>
<td>The docstring of the class</td>
</tr>
<tr>
<td><code>__hash__</code> (self)</td>
<td>Returns hash value (integer) of object</td>
</tr>
<tr>
<td></td>
<td>Dictionary keys and set values must have a <code>__hash__</code> method</td>
</tr>
<tr>
<td><code>__lt__</code> (self, other)</td>
<td>Comparison (less than, &lt;) used by <code>sorted</code> and <code>sort()</code></td>
</tr>
<tr>
<td><code>__init__</code> (self,...)</td>
<td>Class initializer</td>
</tr>
</tbody>
</table>

Class hierarchies

- inheritance
- method overriding
- super
- multiple inheritance
Calling methods of a class

- If an object \textit{obj} of class \textit{C} has a method \textit{method}, then usually you call \textit{obj.method}()

- It is possible to call the method in the class directly using \textit{C.method}, where the object is the first argument

\textit{C.method(obj)}

\texttt{X.py}

\begin{verbatim}
class X:
    def set_x(self, x):
        self.x = x
    def get_x(self):
        return self.x

obj = X()
obj.set_x(42)
print("obj.get_x() =", obj.get_x())
print("obj.x =", obj.x)
print("X.get_x(obj) =", X.get_x(obj))
\end{verbatim}

\texttt{Python shell}

| obj.get_x() = 42 |
| obj.x = 42 |
| X.get_x(obj) = 42 |
# Classes and Objects

## class Person
- set_name(name)
- get_name()
- set_address(address)
- get_address()

## class Student
- set_name(name)
- get_name()
- set_address(address)
- get_address()
- set_id(student_id)
- get_id()
- set_grade(course, grade)
- get_grades()

### Observation: students and employees are persons with additional attributes

<table>
<thead>
<tr>
<th>Student object</th>
</tr>
</thead>
<tbody>
<tr>
<td>name = 'Donald Duck'</td>
</tr>
<tr>
<td>address = 'Duck Steet 13, Duckburg'</td>
</tr>
<tr>
<td>id = '1094'</td>
</tr>
<tr>
<td>grades = {'programming' : 'A' }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Employee object</th>
</tr>
</thead>
<tbody>
<tr>
<td>name = 'Goofy'</td>
</tr>
<tr>
<td>address = 'Clumsy Road 7, Duckburg'</td>
</tr>
<tr>
<td>employer = 'Yarvard University'</td>
</tr>
</tbody>
</table>

---

**Observation:**

Students and employees are persons with additional attributes.
Classes and Objects

**Goal** – avoid redefining the 4 methods below from `person` class again in `student` class

```python
class Person:
    def set_name(self, name):
        self.name = name
    def get_name(self):
        return self.name
    def set_address(self, address):
        self.address = address
    def get_address(self):
        return self.address

class Student:
    def set_name(self, name):
        # Person attributes
    def get_name(self):
        return self.name
    def set_address(self, address):
        # Person attributes
    def get_address(self):
        return self.address

set_id(student_id)
get_id()
set_grade(course, grade)
get_grades()
```
class Student inherits from class Person

class Person is the base class of Student

class Person:
    set_name(name)
    get_name()
    set_address(address)
    get_address()

class Student:
    set_name(name)
    get_name()
    set_address(address)
    get_address()
    set_id(student_id)
    get_id()
    set_grade(course, grade)
    get_grades()
Classes constructors

class Person

set_name(name)
get_name()
set_address(address)
get_address()

class Student(Person):

set_name(name)
get_name()
set_address(address)
get_address()

set_id(student_id)
get_id()
set_grade(course, grade)
get_grades()

person.py

class Person:
    def __init__(self):
        self.name = None
        self.address = None
...

class Student(Person):
    def __init__(self):
        self.id = None
        self.grades = {}
        Person.__init__(self)
...

Notes
1) If Student.__init__ is not defined, then Person.__init__ will be called
2) Student.__init__ must call Person.__init__ to initialize the name and address attributes
super()

**class** Person

```python
set_name(name)
get_name()
set_address(address)
get_address()
```

**class** Student

```python
set_name(name)  # person attributes
get_name()
set_address(address)
get_address()
set_id(student_id)
get_id()
set_grade(course, grade)
get_grades()
```

**person.py**

```python
class Person:
    def __init__(self):
        self.name = None
        self.address = None

    def __init__(self):
        super().__init__()

    def __init__(self):
        super().__init__()

class Student(Person):
    def __init__(self):
        self.id = None
        self.grades = {}
        Person.__init__(self)
        super().__init__()

    def __init__(self):
        super().__init__()

    def __init__(self):
        super().__init__()
```

**Notes**

1) Function `super()` searches for attributes in base class
2) `super` is often a keyword in other OO languages, like Java and C++
3) Note `super().__init__()` does not need `self` as argument
Method search order

```python
class Person:
    def set_name(self, name):
        pass
    def get_name(self):
        pass
    def set_address(self, address):
        pass
    def get_address(self):
        pass

class Student(Person):
    def set_id(self, student_id):
        pass
    def get_id(self):
        pass
    def set_grade(self, course, grade):
        pass
    def get_grades(self):
        pass

student = Student()
student.name = 'Donald Duck'
student.address = 'Duck Street 13, Duckburg'
student.id = '1094'
student.grades = {'programming': 'A'}
```
class Person

set_name(name)
get_name()

set_address(address)
get_address()

class Student(Person)

set_id(student_id)
get_id()

set_grade(course, grade)
get_grades()

class Employee(Person)

set_employer(employer)
get_employer()
Method overriding

In Java one can use the keyword "finally" to prevent any subclass to override a method.
Question – What does `b.f()` print?

Python shell

```python
> class A():
    def f(self):
        print("Af")
        self.g()
    def g(self):
        print("Ag")

> class B(A):
    def g(self):
        print("Bg")

> b = B()
> b.f()
```

a) AttributeError
b) Af Ag

c) Af Bg

d) Don’t know
Undefine methods in superclass?

Python shell

```python
> class A():
    def f(self):
        print("Af")
        self.g()
    def g(self):
        print("Ag")
> class B(A):  
    def g(self):
        print("Bg")
> b = B()
> b.f()
| Af
| Bg
> a = A()
> a.f()
| Af
| AttributeError: 'A' object has no attribute 'g'
```

Method `g` undefined in class `A`; subclasses must implement `g` to be able to call `f` in Java, `A` would have been required to be declared an `abstract class`

can create instance of `A` fails since `g` is not defined in class `A`
Name mangling and inheritance

- The call to `A.__g` in `A.f` forces a call to `__g` to stay within `A`

- Recall that due to name mangling, `__g` is accessible as `A._A__g`
Multiple inheritance

- A class can inherit attributes from multiple classes (in example two)
- When calling a method defined in several ancestor classes, Python executes only one of the these (in the example `say_hello`)
- Which one is determined by the so called "C3 Method Resolution Order" (originating from the Dylan language)

```python
class Alice:
    def say_hello(self):
        print("Alice says hello")
    def say_good_night(self):
        print("Alice says good night")

class Bob:
    def say_hello(self):
        print("Bob says hello")
    def say_good_morning(self):
        print("Bob says good morning")

class X(Alice, Bob): # Multiple inheritance
    def say(self):
        self.say_good_morning()
        self.say_hello() # C3 resolution
        Alice.say_hello(self) # from Alice
        Bob.say_hello(self) # from Bob
        self.say_good_night()
```

```bash
> X().say()
| Bob says good morning
| Alice says hello
| Alice says hello
| Bob says hello
| Alice says good night
```

Raymond Hettinger, *Super considered super!*
Conference talk at PyCon 2015
C3 Method resolution order

- Use `help(class)` to determine the resolution order for the class
- or access the `__mro__` attribute of the class
Question – Who says hello ? Bob says good morning

a) Alice
b) Bob
c) Don’t know
Comparing objects and classes

- `id(obj)` returns a unique identifier for an object (in CPython the memory address)
- `obj1 is obj2` tests if `id(obj1)==id(obj2)`
- `type(obj)` and `obj.__class__` return the class of an object
- `isinstance(object, class)` checks if an object is of a particular class, or a derived subclass
- `issubclass(class1, class2)` checks if `class1` is a subclass of `class2`
is is not for integers, strings, ... and is is not ==

- Only use is on objects!
- Even though isinstance(42, object) and isinstance("abc", object) are true, do not use is on integers and strings!
Comparison of OO in Python, Java and C++

- private, public, .... – in Python everything in an object is public
- class inheritance – core concept in OO programming
  - Python and C++ support multiple inheritance
  - Java only allows single inheritance, but Java ”interfaces” allow for something like multiple inheritance
- Python and C++ allows overloading standard operators (+, *, ...). In Java it is not possible.
- Overloading methods
  - Python extremely dynamic (hard to say anything about the behaviour of a program in general)
  - Java and C++’s type systems allow several methods with same name in a class, where they are distinguished by the type of the arguments, whereas Python allows only one method that can have * and ** arguments
C++ example

- Multiple methods with identical name (`print`)
- The types distinguish the different methods

printing.cpp

```cpp
#include <iostream>
using namespace std;

class MyClass {
public:
    void print(int x) {
        cout << "An integer " << x << endl;
    }
    void print(string s) {
        cout << "A string " << s << endl;
    }
};

main() {
    MyClass C;
    C.print(42);
    C.print("abc");
}
```

printing.py

```python
class MyClass:
    def print(self, value):
        if isinstance(value, int):
            print('An integer', value)
        elif isinstance(value, str):
            print('A string', value)
C = MyClass()
C.print(42)
C.print("abc")
```

Shell

```
| An integer 42 |
| A string abc |
```
Exceptions and file input/output

- try-raise-except-finally
- Exception
- control flow
- file open/read/write
- sys.stdin, sys.stdout, sys.stderr
Exceptions – Error handling and control flow

- **Exceptions** is a widespread technique to handle run-time *errors* / abnormal behaviour (e.g. in Python, Java, C++, JavaScript, C#)

- **Exceptions** can also be used as an *advanced control flow mechanism* (e.g. in Python, Java, JavaScript)

  - *Problem: How to perform a “break” in a recursive function?*
Built-in exceptions
(class hierarchy)

BaseException
  +-- SystemExit
  +-- KeyboardInterrupt
  +-- GeneratorExit
  +-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
      |   +-- FloatingPointError
      |   +-- OverflowError
      |   +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
      |   +-- ModuleNotFoundError
    +-- LookupError
      |   +-- IndexError
      |   +-- KeyError
    +-- MemoryError
    +-- NameError
      |   +-- UnboundLocalError
    +-- TypeError
    +-- ValueError
      |   +-- UnicodeError
      |   |   +-- UnicodeDecodeError
      |   |   +-- UnicodeEncodeError
      |   |   +-- UnicodeTranslateError
    +-- ReferenceError
    +-- RuntimeError
      |   +-- NotImplementedError
      |   +-- RecursionError
    +-- SyntaxError
      |   +-- IndentationError
      |   +-- TabError
    +-- SystemError
    +-- Warning
      |   +-- DeprecationWarning
      |   +-- PendingDeprecationWarning
      |   +-- RuntimeWarning
      |   +-- SyntaxWarning
      |   +-- UserWarning
      |   +-- FutureWarning
      |   +-- ImportWarning
      |   +-- UnicodeWarning
      |   +-- BytesWarning
      |   +-- ResourceWarning
  +-- OSError
    |   +-- BlockingIOError
    |   +-- ChildProcessError
    |   +-- ConnectionError
    |   |   +-- BrokenPipeError
    |   |   +-- ConnectionAbortedError
    |   |   +-- ConnectionRefusedError
    |   |   +-- ConnectionResetError
    |   +-- FileExistsError
    |   +-- FileNotFoundError
    |   +-- InterruptedError
    |   +-- IsADirectoryError
    |   +-- NotADirectoryError
    |   +-- PermissionError
    |   +-- ProcessLookupError
    |   +-- TimeoutError
  +-- ReferenceError
  +-- RuntimeError
    |   +-- NotImplementedError
    |   +-- RecursionError
  +-- SyntaxError
    |   +-- IndentationError
    |   +-- TabError
  +-- SystemError
  +-- Warning
    |   +-- DeprecationWarning
    |   +-- PendingDeprecationWarning
    |   +-- RuntimeWarning
    |   +-- SyntaxWarning
    |   +-- UserWarning
    |   +-- FutureWarning
    |   +-- ImportWarning
    |   +-- UnicodeWarning
    |   +-- BytesWarning
    |   +-- ResourceWarning
Typical built-in exceptions and unhandled behaviour
Catching exceptions – Fractions (I)

```python
fraction1.py
while True:
    numerator = int(input('Numerator = '))
    denominator = int(input('Denominator = '))
    result = numerator / denominator
    print('%s / %s = %s' % (numerator, denominator, result))
```

Python shell

| Numerator = 10 |
| Denominator = 3 |
| 10 / 3 = 3.3333333333333335 |
| Numerator = 20 |
| Denominator = 0 |
| ZeroDivisionError: division by zero |
Catching exceptions – Fractions (II)

```
fraction2.py

while True:
    numerator = int(input('Numerator = '))
    denominator = int(input('Denominator = '))
    try:
        result = numerator / denominator
    except ZeroDivisionError:
        print('cannot divide by zero')
        continue
    print('%s / %s = %s' % (numerator, denominator, result))
```

Python shell

| Numerator = 10 |
| Denominator = 0 |
| cannot divide by zero |
| Numerator = 20 |
| Denominator = 3 |
| 20 / 3 = 6.666666666666667 |
| Numerator = 42x |
| **ValueError**: invalid literal for int() with base 10: '42x' |
Catching exceptions – Fractions (III)

```python
fraction3.py
while True:
    try:
        numerator = int(input('Numerator = '))
        denominator = int(input('Denominator = '))
    except ValueError:
        print('input not a valid integer')
        continue
    try:
        result = numerator / denominator
    except ZeroDivisionError:
        print('cannot divide by zero')
        continue
    print('%s / %s = %s' % (numerator, denominator, result))
```

Python shell
```
| Numerator = 5
| Denominator = 2
| input not a valid integer
| Numerator = 5
| Denominator = 2
| 5 / 2 = 2.5
```
while True:
    try:
        numerator = int(input('Numerator = '))
        denominator = int(input('Denominator = '))
    except ValueError:
        print('input not a valid integer')
        continue
    try:
        result = numerator / denominator
        print('%s / %s = %s' % (numerator, denominator, result))
    except ZeroDivisionError:
        print('cannot divide by zero')
Catching exceptions – Fractions (IV)

```python
catch exceptions

```
```python
fraction4.py

```python
while True:
    try:
        numerator = int(input('Numerator = '))
        denominator = int(input('Denominator = '))
        result = numerator / denominator
        print('%s / %s = %s' % (numerator, denominator, result))
    except ValueError:
        print('input not a valid integer')
    except ZeroDivisionError:
        print('cannot divide by zero')
```
Keyboard interrupt (Ctrl-c)

- throws KeyboardInterrupt exception

```python
infinite-loop1.py
print('starting infinite loop')
x = 0
while True:
    x = x + 1
print('done (x = %s)' % x)
input('type enter to exit')
```

```python
infinite-loop2.py
print('starting infinite loop')
try:
x = 0
while True:
    x = x + 1
except KeyboardInterrupt:
    pass
print('done (x = %s)' % x)
input('type enter to exit')
```

Python shell
```
starting infinite loop
Traceback (most recent call last):
  File 'infinite-loop1.py', line 4, in <module>
x = x + 1
KeyboardInterrupt
```

Python shell
```
starting infinite loop
done (x = 23890363)  # Ctrl-c
type enter to exit
```
Keyboard interrupt (Ctrl-c)

- Be aware that you likely would like to leave the Ctrl-c generated `KeyboardInterrupt` exception unhandled, except when stated explicitly

```
read-int1.py
while True:
    try:
        x = int(input('An integer: '))
        break
    except ValueError:  # only ValueError
        continue
print('The value is:', x)
```

Python shell

```
An integer: Ctrl-c
KeyboardInterrupt
```

```
read-int2.py
while True:
    try:
        x = int(input('An integer: '))
        break
    except:  # all exceptions
        continue
print('The value is:', x)
```

Python shell

```
An integer: Ctrl-c
An integer: Ctrl-c
An integer: 
```

- (left) `KeyboardInterrupt` is unhandled (right) it is handled (intentionally?)

```
catches KeyboardInterrupt
```

---

Keyboard interrupt (Ctrl-c)

- Be aware that you likely would like to leave the Ctrl-c generated `KeyboardInterrupt` exception unhandled, except when stated explicitly

```
read-int1.py
while True:
    try:
        x = int(input('An integer: '))
        break
    except ValueError:  # only ValueError
        continue
print('The value is:', x)
```

Python shell

```
An integer: Ctrl-c
KeyboardInterrupt
```

```
read-int2.py
while True:
    try:
        x = int(input('An integer: '))
        break
    except:  # all exceptions
        continue
print('The value is:', x)
```

Python shell

```
An integer: Ctrl-c
An integer: Ctrl-c
An integer: 
```

- (left) `KeyboardInterrupt` is unhandled (right) it is handled (intentionally?)

```
catches KeyboardInterrupt
```
Exception class hierarchy

```python
except-twice1.py
try:
    L[4]
except IndexError:  # must be before Exception
    print('IndexError')
except Exception:
    print('Fall back exception handler')
except-twice2.py
try:
    L[4]
except Exception:  # and subclasses of Exception
    print('Fall back exception handler')
except IndexError:
    print('IndexError')  # unreachable
```
try statement syntax

```python
try:
    code

    except ExceptionType1:
        code  # executed if raised exception instanceof
            # ExceptionType1 (or subclass of ExceptionType1)

    except ExceptionType2:
        code  # executed if exception type matches and none of
            # the previous except statements matched

...

else:
    code  # only executed if no exception was raised

finally:
    code  # always executed independent of exceptions
        # typically used to clean up (like closing files)
```
except variations

except: # catch all exceptions

except ExceptionType: # only catch exceptions of class ExceptionType
    # or subclasses of ExceptionType

except (ExceptionType₁, ExceptionType₂, ..., ExceptionTypeₖ): # catch any of k classes (and subclasses)

except ExceptionType as e:
    # catch exception and assign exception object to e
    # e.args contains arguments to the raised exception
Raising exceptions

- An exception is raised (or thrown) using one of the following (the first being an alias for the second):
  
  ```
  raise ExceptionType
  raise ExceptionType()
  raise ExceptionType(args)
  ```

```python
abstract.py

class A():
    def f(self):
        print('f')
        self.g()

    def g(self):
        raise NotImplementedError

class B(A):
    def g(self):
        print('g')

Python shell

$ B().f()
   f
   g

$ A().f()
   f
   NotImplementedError
```
User exceptions

- New exception types are created using `class inheritance` from an existing exception type (possibly defining `__init__`)
User exception with argument in recursion

- Escape from recursion by raising exception
- Pass result as argument to exception
- Unpack .args tuple from result caught by except

```python
class SolutionFound(Exception):
    pass

def apx_tree_search(x, tree):
    def search(x, t):
        if isinstance(t, tuple):
            for child in t:
                search(x, child)
        elif abs(x - t) < 1:  # approximate match
            raise SolutionFound(t)
    try:
        search(x, tree)
    except SolutionFound as e:
        result, = e.args  # e.args is a tuple
        print('search for', x, 'found', result)
    else:
        print('search for', x, 'unsuccessful')

tree = ((3.2, 2.1), 5.6, (7.8, (9.3, 6.5)))
apx_tree_search(4.3, tree)
apx_tree_search(5.9, tree)
```

Python shell

| search for 4.3 unsuccessful |
| search for 5.9 found 5.6 |
3 ways to read lines from a file

Steps

1. Open file using `open`
2. Read lines from file using
   a) `filehandler.readline`
   b) `filehandler.readlines`
   c) for line in `filehandler`:
3. Close file using `close`

---

open ('filename.txt') assumes the file to be in the same folder as your Python program, but you can also provide a full path
open('c:/Users/gerth/Documents/filename.txt')
3 ways to write lines to a file

- Opening file:
  \( \texttt{open(filename, mode)} \)
  where \( \texttt{mode} \) is a string, either \( \texttt{w} \) for opening a new (or truncating an existing file) and \( \texttt{a} \) for appending to an existing file

- Write single string:
  \( \texttt{filehandle.write(string)} \)
  Returns the number of characters written

- Write list of strings \( \texttt{strings} \):
  \( \texttt{filehandle.writelines(list)} \)

- Newlines (\( \texttt{\'\n\'} \)) must be written explicitly

- \( \texttt{print} \) can take an optional \( \texttt{file} \) argument

```
write-file.py

f = open('output-file.txt', 'w')
f.write('Text 1\n')
f.writelines(['Text 2\n', 'Text 3 '])
f.close()

g = open('output-file.txt', 'a')
print('Text 4', file=g)
g.writelines(['Text 5 ', 'Text 6'])
g.close()
```

```
output-file.txt
Text 1
Text 2
Text 3 Text 4
Text 5 Text 6
```
Exceptions while dealing with files

- When dealing with files one should be prepared to handle errors / raised exceptions, e.g. `FileNotFoundError`

```python
reading-file4.py

try:
    f = open('reading-file4.py')
except FileNotFoundError:
    print('Could not open file')
else:
    try:
        for line in f:
            print('> ', line[:-1])
    finally:
        f.close()
```
Opening files using `with`

- The Python keyword `with` allows to create a *context manager* for handling files.
- *Filehandle will automatically be closed, also when exceptions occur.*
- Under the hood: filehandles returned by `open()` support `__enter__()` and `__exit__()` methods.

```python
reading-file5.py

with open('reading-file5.py') as f:
    for line in f:
        print('> ', line[:-1])
```

- `f` = result of calling `__enter__()` on result of `open` expression, which is the file handle.
Does a file exist?

- Module `os.path` contains a method `isfile` to check if a file exists.

```python
checking-files.py
import os.path

filename = input('Filename: ')  
if os.path.isfile(filename):
    print('file exists')
else:
    print('file does not exists')
```

docs.python.org/3/library/os.path.html
Module *sys* contains the three standard file handles:

- `sys.stdin` (used by the `input` function)
- `sys.stdout` (used by the `print` function)
- `sys.stderr` (error output from the Python interpreter)

### sys-test.py

```python
import sys
sys.stdout.write('Input an integer: 
)x = int(sys.stdin.readline())
sys.stdout.write('%s square is %s' % (x, x**2))
```

### Python shell

```
Input an integer: 10
10 square is 100
```
print(..., file=\texttt{output file})

\begin{center}
\texttt{sys-print-file.py}
\end{center}

```python
import sys
def complicated_function(file):
    print('Hello world', file=file)  # print to file or STDOUT

while True:
    file_name = input('Output file (empty for STDOUT): ')

    if file_name == '':
        file = sys.stdout
        break
    else:
        try:
            file = open(file_name, 'w')
        except Exception:
            pass
        break

complicated_function(file)

if file != sys.stdout:
    file.close()
```
PEP8 on exceptions

- For all try/except clauses, limit the try clause to the absolute minimum amount of code necessary.
- The class naming convention applies (CapWords).
- Use the suffix "Error" on your exception names (if the exception actually is an error).
- A bare `except:` clause will catch `SystemExit` and `KeyboardInterrupt` exceptions, making it harder to interrupt a program with Control-C, and can disguise other problems. If you want to catch all exceptions that signal program errors, use `except Exception:`.
Performance of scanning a file

- Python can efficiently scan through quite big files

<table>
<thead>
<tr>
<th>File</th>
<th>Size</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom_chem_shift.csv</td>
<td>≈ 750 MB</td>
<td>≈ 8 sec</td>
</tr>
<tr>
<td>cano.txt</td>
<td>≈ 3.7 MB</td>
<td>≈ 0.1 sec</td>
</tr>
</tbody>
</table>

The first search finds all lines related to ThrB12-DKP-insulin (Entry ID 6203) in a chemical database available from [www.bmrb.wisc.edu](http://www.bmrb.wisc.edu)

The second search finds all occurrences of “Germany” in Conan Doyle's complete Sherlock Holmes available at [sherlock-holmes.es](http://sherlock-holmes.es)

---

```python
from time import time

for filename, query in [('Atom_chem_shift.csv', ',6203,'), ('cano.txt', 'Germany')]:
    count = 0
    matches = []
    start = time()
    with open(filename) as f:
        for i, line in enumerate(f, start=1):
            count += 1
            if query in line:
                matches.append((i, line))
    end = time()
    for i, line in matches:
        print(i, ':', line, end='')
    print('Duration:', end - start)
    print(len(matches), 'of', count, 'lines match')
```

---

**File scanning results**

<table>
<thead>
<tr>
<th>Duration</th>
<th>Lines match</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.760039329528809</td>
<td>329 of 9758361 lines match</td>
</tr>
<tr>
<td>0.0770065784454357</td>
<td>4 of 76764 lines match</td>
</tr>
</tbody>
</table>
```python
class Sudoku:
    def __init__(self, puzzle):
        self.puzzle = puzzle

    def solve(self):
        def find_free():
            for i in range(9):
                for j in range(9):
                    if self.puzzle[i][j] == 0:
                        return (i, j)
            return None

        def unused(i, j):
            i_, j_ = i // 3 * 3, j // 3 * 3
            cells = {(i, k) for k in range(9)}
            cells |= {(k, j) for k in range(9)}
            cells |= {(i, j) for i in range(i_, i_ + 3)
                      for j in range(j_, j_ + 3)}
            return set(range(1, 10)) - {self.puzzle[i][j] for i, j in cells}

        class SolutionFound(Exception):
            pass

        def recursive_solve():
            cell = find_free()
            if not cell:
                raise SolutionFound
            i, j = cell
            for value in unused(i, j):
                self.puzzle[i][j] = value
                recursive_solve()
            self.puzzle[i][j] = 0

        try:
            recursive_solve()
        except SolutionFound:
            pass

    def print(self):
        for i, row in enumerate(self.puzzle):
            cells = [' %s ' % c if c else ' . ' for c in row]
            print('|'.join([''.join(cells[j:j+3]) for j in (0,3,6)]))
            if i in (2,5):
                print('---------+---------+---------')
        with open('sudoku.txt') as f:
            A = Sudoku([[int(x) for x in line.strip()] for line in f])
            A.solve()
            A.print()
```

```text
sudoku.py (continued)
```
Documentation, testing and debugging

- docstring
- defensive programming
- assert
- test driven development
- assertions
- testing
- unittest
- debugger
- static type checking (mypy)
Ensuring good quality code?

**Goal**
- Develop programs that work correctly
- Tools and techniques

**Design phase**
- Idea

**Development**
- Coding
- Fix bug
- Testing
- Find bug

**Usage**
- User runs program
  - (hopefully) correct program
  - runs forever / crash / incorrect output / explosion / ...
What is good code?

- **Readability**
  - well-structured
  - documentation
  - comments
  - follow some standard structure (easy to recognize, follow [PEP8 Style Guide](https://www.python.org/dev/peps/pep-0008/))

- **Correctness**
  - outputs the correct answer on valid input
  - eventually stops with an answer on valid input (should not go in infinite loop)

- **Reusable...**
Why?

Documentation
- specification of functionality
- docstring – for users of the code
  - modules
  - methods
  - classes
- comments – for readers of the code

Testing
- Correct implementation?
- Try to predict behavior on unknown input?
- Performance guarantees?

Debugging
- Where is the bug?

“Program testing can be used to show the presence of bugs, but never to show their absence” – Edsger W. Dijkstra
Built-in exceptions
(class hierarchy)

BaseException
  +-- SystemExit
  +-- KeyboardInterrupt
  +-- GeneratorExit
  +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
          |    +-- FloatingPointError
          |    +-- OverflowError
          |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
          |    +-- ModuleNotFoundError
      +-- LookupError
          |    +-- IndexError
          |    +-- KeyError
      +-- MemoryError
      +-- NameError
          |    +-- UnboundLocalError
      +-- TypeError
          |    +-- UnicodeError
              |    +-- UnicodeDecodeError
              |    +-- UnicodeEncodeError
              |    +-- UnicodeTranslateError
      +-- ValueError
      +-- ReferenceError
      +-- RuntimeError
          |    +-- NotImplementedError
          |    +-- RecursionError
      +-- SyntaxError
          |    +-- IndentationError
          |    +-- TabError
      +-- SystemError
      +-- Warning
          |    +-- DeprecationWarning
          |    +-- PendingDeprecationWarning
          |    +-- RuntimeWarning
          |    +-- SyntaxWarning
          |    +-- UserWarning
          |    +-- FutureWarning
          |    +-- ImportWarning
          |    +-- UnicodeWarning
          |    +-- BytesWarning
          |    +-- ResourceWarning

docs.python.org/3/library/exceptions.html
Testing for unexpected behaviour?

**infinite-recursion1.py**

```python
def f(depth):
    f(depth + 1)  # infinite recursion
f(0)
```

Python shell

```
RecursionError: maximum recursion depth exceeded
```

**infinite-recursion2.py**

```python
def f(depth):
    if depth > 100:
        print("runaway recursion???")
        raise SystemExit  # raise built-in exception
    f(depth + 1)
f(0)
```

Python shell

```
runaway recursion???
```

**infinite-recursion3.py**

```python
import sys

def f(depth):
    if depth > 100:
        print("runaway recursion???")
        sys.exit()  # system function
    f(depth + 1)
f(0)
```

Python shell

```
runaway recursion???
```

- let the program eventually fail
- check and raise exceptions
- check and call `sys.exit`
Catching unexpected behaviour – **assert**

- **keyword** `assert` checks if boolean expression is true, if not, raises exception `AssertionError`

- optional second parameter passed to the constructor of the exception

---

### infinite-recursion4.py

```python
def f(depth):
    assert depth <= 100  # raise exception if False
    f(depth + 1)
f(0)
```

**Python shell**

```bash
File "...\infinite-recursion4.py", line 2, in f
    assert depth <= 100
AssertionError
```

### infinite-recursion5.py

```python
def f(depth):
    assert depth <= 100, "runaway recursion???
    f(depth + 1)
f(0)
```

**Python shell**

```bash
File "...\infinite-recursion5.py", line 2, in f
    assert depth <= 100
AssertionError
```

### infinite-recursion6.py

```python
def f(depth):
    if not depth <= 100:
        raise AssertionError("runaway recursion???")
    f(depth + 1)
f(0)
```

**Python shell**

```bash
File "...\infinite-recursion6.py", line 3, in f
    raise AssertionError("runaway recursion???")
AssertionError: runaway recursion???
```
Disabling `assert` statements

- `assert` statements are good to help check correctness of program – but can slow down program.

- Invoking Python with option `--O` disables all assertions (by setting `__debug__` to `False`).

[ docs.python.org/3/reference/simple_stmts.html#assert ]
Example

\[ \sqrt{x} \]
First try... (seriously, the bugs were not on purpose)

```python
# intsqrt_buggy.py

def int_sqrt(x):
    low = 0
    high = x
    while low < high - 1:
        mid = (low + high) / 2
        if mid ** 2 <= x:
            low = mid
        else:
            high = mid
    return low

# Python shell
> int_sqrt(10)
| 3.125  # 3.125 ** 2 = 9.765625
> int_sqrt(-10)
| 0  # what should the answer be ?
```
Let us add a specification...

```python
int_sqrt.py

```def int_sqrt(x):
    
    
    """Compute the integer square root of an integer x.
    
    Assumes that x is an integer and x >= 0
    Returns integer floor(sqrt(x))"
    ...

Python shell

```python
> help(int_sqrt)
| Help on function int_sqrt in module __main__:

| int_sqrt(x)
| Compute the integer square root of an integer x.
| Assumes that x is an integer and x >= 0
| Returns integer floor(sqrt(x))

```
Let us check input requirements...

```python
def int_sqrt(x):
    """Compute the integer square root of an integer x.

    Assumes that x is an integer and x >= 0
    Returns integer floor(sqrt(x))"
    assert isinstance(x, int)
    assert 0 <= x
...
```

- doing explicit checks for valid input arguments is part of **defensive programming** and helps spotting errors early

(instead of continuing using likely wrong values... resulting in a final meaningless error)
Let us check if output correct...

```python
int_sqrt.py

def int_sqrt(x):
    """Compute the integer square root of an integer x.
    Assumes that x is an integer and x >= 0
    Returns integer floor(sqrt(x))""

    assert isinstance(x, int)
    assert 0 <= x
    ...  
    assert isinstance(result, int)  
    assert result ** 2 <= x < (result + 1) ** 2
    return result

Python shell

> int_sqrt(10)
|   File "...\int_sqrt.py", line 20, in int_sqrt
|     assert isinstance(result, int)
|     AssertionError
```

- output check identifies the error
  ```
  mid = (low+high) / 2
  ```
- should have been
  ```
  mid = (low+high) // 2
  ```
- The output check helps us to ensure that function specifications are satisfied in applications
Let us test some input values...

```python
def int_sqrt(x):
    ...

assert int_sqrt(0) == 0
assert int_sqrt(1) == 1
assert int_sqrt(2) == 1
assert int_sqrt(3) == 1
assert int_sqrt(4) == 2
assert int_sqrt(5) == 2
assert int_sqrt(200) == 14
```

- test identifies wrong output for $x = 1$
Let us check progress of algorithm...

```python
... lows = 0, x
while low < high - 1:  # low <= floor(sqrt(x)) < high
    assert low ** 2 <= x < high ** 2
    mid = (low + high) // 2
    if mid ** 2 <= x:
        low = mid
    else:
        high = mid
result = low
...
```

- test identifies wrong output for $x = 1$
- but invariant apparently correct ???
- problem
  
  ```python
  low == result == 0
  high == 1
  ```
  
  implies loop never entered
- output check identifies the error
  ```
  high = x
  ```
- should have been
  ```
  high = x + 1
  ```
Final program

We have used assertions to:

- Test if input arguments / usage is valid (defensive programming)
- Test if computed result is correct
- Test if an internal invariant in the computation is satisfied
- Perform a final test for a set of test cases (should be run whenever we change anything in the implementation)
Which checks would you add to the below code?

```python
def binary_search(x, L):
    """Binary search for x in sorted list L

    Assumes x is an integer, and L a non-decreasing list of integers

    Returns index i, -1 <= i < len(L), where L[i] <= x < L[i+1],
    assuming L[-1] = -infty and L[len(L)] = +infty"

    low, high = -1, len(L)
    while low + 1 < high:
        mid = (low + high) // 2
        if x < L[mid]:
            high = mid
        else:
            low = mid
    result = low
    return result
```
def binary_search(x, L):
    """Binary search for x in sorted list L
    Assumes x is an integer, and L a non-decreasing list of integers
    Returns index i, -1 <= i < len(L), where L[i] <= x < L[i+1],
    assuming L[-1] = -infty and L[len(L)] = +infty"
    assert isinstance(x, int)
    assert isinstance(L, list)
    assert all([isinstance(e, int) for e in L])
    assert all([L[i] <= L[i + 1] for i in range(len(L) - 1)])
    low, high = -1, len(L)
    while low + 1 < high:
        # L[low] <= x < L[high]
        mid = (low + high) // 2
        assert isinstance(L[mid], int)
        assert (low == -1 or L[low] <= L[mid]) and (high == len(L) or L[mid] <= L[high])
        if x < L[mid]:
            high = mid
        else:
            low = mid
    result = low
    assert (isinstance(result, int) and -1 <= result < len(L) and
            ((result == -1 and (len(L) == 0 or x < L[0])) or
             (result == len(L) - 1 and x >= L[-1]) or
             (0 <= result < len(L) - 1 and L[result] <= x < L[result + 1])))
    return result

assert binary_search(42, []) == -1
assert binary_search(42, [7]) == 0
assert binary_search(7, [42]) == -1
assert binary_search(7, [42, 42, 42]) == -1
assert binary_search(42, [7, 7, 7]) == 2
assert binary_search(42, [7, 7, 7, 56, 81]) == 2
assert binary_search(8, [1, 3, 5, 7, 9]) == 3

① Verifying if L is a sorted list of integers can slow down the program significantly
② Alternative is to only verify if the part of L visited is a sorted subsequence

Verifying if L is a sorted list of integers can slow down the program significantly
Alternative is to only verify if the part of L visited is a sorted subsequence
Testing – how?

- Run set of test cases
  - test all cases in input/output specification (black box testing)
  - test all special cases (black box testing)
  - set of tests should force all lines of code to be tested (glass box testing)

- Visual test

- Automatic testing
  - Systematically / randomly generate input instances
  - Create function to validate if output is correct (hopefully easier than finding the solution)

- Formal verification
  - Use computer programs to do formal proofs of correctness, like using Coq
Visual testing – Convex hull computation

Correct

Bug!
(not convex)
### doctest

- **Python module**
- **Test instances (pairs of input and corresponding output) are written in the docstrings, formatted as in an interactive Python session**

```python
import doctest
doctest.testmod((verbose=True)
```

<table>
<thead>
<tr>
<th>binary-search-doctest.py</th>
<th>Python shell</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>def binary_search(x, L):</code></td>
<td>Trying:</td>
</tr>
<tr>
<td>&quot;&quot;&quot;Binary search for x in sorted list L&quot;&quot;&quot;</td>
<td>binary_search(42, [])</td>
</tr>
<tr>
<td></td>
<td>Expecting:</td>
</tr>
<tr>
<td></td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>ok</td>
</tr>
<tr>
<td></td>
<td>Trying:</td>
</tr>
<tr>
<td></td>
<td>binary_search(42, [7])</td>
</tr>
<tr>
<td></td>
<td>Expecting:</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>ok</td>
</tr>
<tr>
<td></td>
<td>Trying:</td>
</tr>
<tr>
<td></td>
<td>binary_search(42, [7,7,7,56,81])</td>
</tr>
<tr>
<td></td>
<td>Expecting:</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>ok</td>
</tr>
<tr>
<td></td>
<td>Trying:</td>
</tr>
<tr>
<td></td>
<td>binary_search(8, [1,3,5,7,9])</td>
</tr>
<tr>
<td></td>
<td>Expecting:</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>ok</td>
</tr>
<tr>
<td></td>
<td>1 items had no tests:</td>
</tr>
<tr>
<td></td>
<td><strong>main</strong></td>
</tr>
<tr>
<td></td>
<td>1 items passed all tests:</td>
</tr>
<tr>
<td></td>
<td><strong>main</strong>.binary_search</td>
</tr>
<tr>
<td></td>
<td>4 tests in <strong>main</strong>.binary_search</td>
</tr>
<tr>
<td></td>
<td>4 tests in 2 items.</td>
</tr>
<tr>
<td></td>
<td>4 passed and 0 failed.</td>
</tr>
<tr>
<td></td>
<td>Test passed.</td>
</tr>
</tbody>
</table>
```
Run all tests stored in functions prefixed by test or test prefixed test methods inside Test prefixed test classes

pip install pytest

Run the pytest program from a shell

pytest.org
**unittest**

- Python module
- A comprehensive **object-oriented test framework**, inspired by the corresponding JUnit test framework for Java

```python
import unittest

class TestBinarySearch(unittest.TestCase):
    def test_search(self):
        self.assertEqual(binary_search(42, []), -1)
        self.assertEqual(binary_search(42, [7]), 0)
        self.assertEqual(binary_search(42, [7,7,7,56,81]), 2)
        self.assertEqual(binary_search(8, [1,3,5,7,9]), 3)

    def test_types(self):
        self.assertRaises(TypeError, binary_search, 5, ['a', 'b', 'c'])

unittest.main(verbosity=2)
```

Python shell
```
Python shell |
<p>| test_search (<strong>main</strong>.TestBinarySearch) ... ok |</p>
<table>
<thead>
<tr>
<th>test_types (<strong>main</strong>.TestBinarySearch) ... ok</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ran 2 tests in 0.051s</td>
</tr>
<tr>
<td>OK</td>
</tr>
</tbody>
</table>
```

[docs.python.org/3/library/unittest.html](https://docs.python.org/3/library/unittest.html)
Debugger (IDLE)

- When an exception has stopped the program, you can examine the state of the variables using **Debug > Stack Viewer** in the Python shell.
Stepping through a program (IDLE debugger)

- **Debug > Debugger** in the Python shell opens Debug Control window
- **Right click** on a code line in editor to set a “breakpoint” in your code
- **Debug Control**: Go → run until next breakpoint is encountered; Step → execute one line of code; Over → run function call without details; Out → finish current function call; Quit → Stop program;
Concluding remarks

- Simple debugging: add print statements
- **Test driven development** → Strategy for code development, where tests are written before the code
- **Defensive programming** → add tests (assertions) to check if input/arguments are valid according to specification
- When designing tests, ensure **coverage** (the set of test cases should make sure all code lines get executed)
- **Python testing frameworks:** doctest, unittest, pytest, ...
Mypy – a static type checker for Python

- **Static type checking** tries to analyze a program for potential type errors without executing the program.

- **Installing:**
  
  ```
  pip install mypy
  ```

- Running Python will cause an error during execution, whereas using mypy the error will be found without executing the program.

- Standard (and required) in statically typed languages like Java, C, C++

```python
mypy-simple.py
print("start")
print(42 + "abc")  # error
print("end")
```

Shell

```
> python mypy-simple.py
| start
| TypeError: unsupported operand type(s) for +: 'int' and 'str'
> mypy mypy-simple.py
| mypy-simple.py:2: error: Unsupported operand types for + ("int" and "str")
```

[pep484](http://mypy-lang.org/PEP-484-Types.html)
Type hints (PEP 484)

- Python allows type hints in programs
- They are ignored at run-time by Python, but useful for static type analysis (mypy)
- Syntax
  
  ```python
  variable : type  
  variable : type = value
  ```

```python
mypy-basic-types.py

x : int # type hint
x = 42
x = "abc" # type error

y : int = 42 # type hint
y = "abc" # type error

z = 42
z = "abc" # type changed from int to str

print(x, y, z)
```

Shell

```
$ python mypy-basic-types.py
| abc abc abc

$ mypy mypy-basic-types.py
| mypy-basic-types.py:3: error: Incompatible types in assignment (expression has type "str", variable has type "int")
| mypy-basic-types.py:6: error: ...
| mypy-basic-types.py:9: error: ...
```
Type hints – functions

Syntax

```python
def name(variable : type, ...) -> return type:
```

<table>
<thead>
<tr>
<th>mypy-function.py</th>
<th>Shell</th>
</tr>
</thead>
</table>
| def f(x: int, units: str) -> str:  
  return str(x) + " " + units  
| > python mypy-function.py  
  | 3 cm  
  | one meter  
  | 3 cm  
  | one meter  
| def g(x, units: str) -> str:  
  return str(x) + " " + units  
| print(f(3, "cm"))  
| print(g(3, "cm"))  
| print(f("one", "meter"))  
| print(g("one", "meter"))  
| > mypy mypy-function.py  
| mypy-function.py:8: error: Argument 1 to "f" has incompatible type "str"; expected "int" |

- Note: for functions and methods `function.__annotations__` is a dictionary with the annotation
Type hints – objects

```python
mypy-classes.py

class A:
    pass

class B(A):
    pass

class C:
    pass

a : A
b : B
c : C

a = A() # valid, B subclass of A
a = B() # valid, B subclass of A
a = C() # error
b = A() # error
b = B() # error
b = C() # error
c = A() # error
c = B() # error
c = C() # error

Shell

> mypy mypy-classes.py
  mypy-classes.py:15: error: Incompatible types in assignment (expression has type "C", variable has type "A")
  mypy-classes.py:16: error: Incompatible types in assignment (expression has type "A", variable has type "B")
  mypy-classes.py:18: error: Incompatible types in assignment (expression has type "C", variable has type "B")
  mypy-classes.py:19: error: Incompatible types in assignment (expression has type "A", variable has type "C")
  mypy-classes.py:20: error: Incompatible types in assignment (expression has type "B", variable has type "C")
```
More type hints... see PEP 484 for even more...

```python
from typing import Mapping, Set, List, Tuple, Union, Optional

S : Set = {} # error {} dictionary
S2 : Set[int] = {1, 2, "abc"} # error "abc" is not int
D : Mapping[int, int] = {1: 42, 'a': 1} # error 'a' is not int
T : Tuple[int, str] = (42, 7) # error 7 is not str
L : List[Union[int, str]] = [42, 'a', None] # list can only contain int and str
L2 : List[Optional[str]] = ['abc', None, 42] # list can only contain str og None
```

Shell

```sh
> mypy mypy-function.py
```

```
mypy-typing.py:3: error: Incompatible types in assignment (expression has type "Dict[<nothing>, <nothing>]", variable has type "Set[<typing.Any>]")
mypy-typing.py:4: error: Argument 3 to <set> has incompatible type "str"; expected "int"
mypy-typing.py:5: error: Dict entry 1 has incompatible type "str": "int"; expected "int": "int"
mypy-typing.py:6: error: Incompatible types in assignment (expression has type "Tuple[<typing.int, <typing.str>]", variable has type "Tuple[<typing.int, <typing.str>]")
mypy-typing.py:7: error: List item 2 has incompatible type "None"; expected "Union[<typing.int, <typing.str>]"
mypy-typing.py:8: error: List item 2 has incompatible type "int"; expected "Optional[<typing.str>]"
```

PEP 484 - Type Hints
Decorators

- @

www.python.org/dev/peps/pep-0318/
### Course Overview

<table>
<thead>
<tr>
<th>Basic programming</th>
<th>Advanced / specific python Libraries &amp; applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction to Python</td>
<td>10. Functions as objects</td>
</tr>
<tr>
<td>2. Python basics / if</td>
<td>11. Object oriented programming</td>
</tr>
<tr>
<td>9. Recursion and Iteration</td>
<td>18. Multi-dimensional data</td>
</tr>
</tbody>
</table>

**10 handins**

**1 final project (last 1 month)**
Python decorators are just syntactic sugar

,pie-decorator' syntax

dec1, dec2, ... are functions (decorators) taking a function as an argument
and returning a new function

Note: decorators are listed bottom up in order of execution
Recap functions

### Recap

1. **Functions**
   - $5 + 3 = 8$
   - $x + y$

2. **List**
   - `sorted(list)`
   - `['defg', 'ij', 'abc']`
   - `len`
   - `list`
   - `key function`

3. **Sorted List**
   - `sorted(list)`
   - `['ij', 'abc', 'defg']`

4. **Decorator**
   - `decorator`
   - `original function`
   - `decorated function`
Contrived example: Plus one (I-II)

Assume we *always* need to call `plus_one` on the result of `square` and `cube` (don’t ask why!)

We could call `plus_one` inside functions (but could be more return statements in functions)
Contrived example: Plus one (III-IV)

```python
# plus_one3.py

def plus_one(x):
    return x + 1

def square(x):
    return x ** 2

def cube(x):
    return x ** 3

square_original = square
cube_original = cube

square = lambda x: plus_one(square_original(x))
cube = lambda x: plus_one(cube_original(x))

print(square(5))
print(cube(5))
```

Python shell

```
26
126
```

Overwrite `square` and `cube` with decorated versions

```python
# plus_one4.py

def plus_one(x):
    return x + 1

def plus_one_decorator(f):
    return lambda x: plus_one(f(x))

def square(x):
    return x ** 2

def cube(x):
    return x ** 3

square = plus_one_decorator(square)
cube = plus_one_decorator(cube)

print(square(5))
print(cube(5))
```

Python shell

```
26
126
```

Create a decorator function `plus_one_decorator`
Contrived example: Plus one (V-VI)

```python
plus_one5.py

def plus_one(x):
    return x + 1
def plus_one_decorator(f):
    return lambda x: plus_one(f(x))
@plus_one_decorator
def square(x):
    return x ** 2
@plus_one_decorator
def cube(x):
    return x ** 3
print(square(5))
print(cube(5))
```

```python
plus_one6.py

def plus_one_decorator(f):
    def plus_one(x):
        return f(x) + 1
    return plus_one
@plus_one_decorator
def square(x):
    return x ** 2
@plus_one_decorator
def cube(x):
    return x ** 3
print(square(5))
print(cube(5))
```

Python shell

| 26 | 126 |

Use Python decorator syntax

Create local function instead of using lambda
Contrived example : Plus one (VII)

A function can have an arbitrary number of decorators (also the same repeated)

Decorators are listed bottom up in order of execution
Handling arguments

"wrapper" is a common name for the function returned by a decorator

@run_twice
def hello_world():
    print("Hello world")

hello_world()

Python shell
| Hello world
| Hello world

run_twice2.py

def run_twice(f):
    def wrapper(*args):
        f(*args)
        f(*args)
    return wrapper

@run_twice
def hello_world():
    print("Hello world")

@run_twice
def hello(txt):
    print("Hello", txt)

hello_world()
hello("Mars")

Python shell
| Hello world
| Hello world
| Hello Mars
| Hello Mars

args holds the arguments in a tuple given to the function to be decorated
Question – What does the decorated program print?

```python
# decorator_quizz.py

def double(f):
    def wrapper(*args):
        return 2 * f(*args)
    return wrapper

def add_three(f):
    def wrapper(*args):
        return 3 + f(*args)
    return wrapper

@double
@add_three
def seven():
    return 7

print(seven())
```

- 7
- 10
- 14
- 17
- 20
- Don’t know
Example: Enforcing argument types

- Defining decorators can be (slightly) complicated
- Using decorators is easy
Decorators can take arguments

Python
@dec(argA, argB, ...)
def func(arg1, arg2, ...):
    pass

Python
def func(arg1, arg2, ...):
    pass
func = dec(argA, argB, ...) (func)

\[\text{dec} \text{ is a function (decorator) that takes a list of arguments and returns a function (to decorate } \text{func} \text{) that takes a function as an argument and returns a new function}\]
Example: Generic type enforcing

```python
def enforce_types(*decorator_args):
    def decorator(f):
        def wrapper(*args):
            assert len(args) == len(decorator_args),
            ("got %s arguments, expected %s" % (len(args), len(decorator_args)))
            assert all([isinstance(x, t) for x, t in zip(args, decorator_args)]),
            "unexpected types"
            return f(*args)
        return wrapper
    return decorator

@example_types(str, int)  # decorator with arguments
def print_repeated(txt, n):
    print(txt * n)
print_repeated("Hello ", 3)
print_repeated("Hello ", "world")
```
Example: A timer decorator

```python
import time

def time_it(f):
    def wrapper(*args, **kwargs):
        t_start = time.time()
        result = f(*args, **kwargs)
        t_end = time.time()
        t = t_end - t_start
        print("%s took %.2f sec" % (f.__name__, t))
        return result

    return wrapper

@time_it
def slow_function(n):
    sum_ = 0
    for x in range(n):
        sum_ += x
    print("The sum is:", sum_)

for i in range(6):
    slow_function(1_000_000 * 2**i)
```

Python shell

<table>
<thead>
<tr>
<th>The sum is: 499999500000</th>
<th>slow_function took 0.27 sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>The sum is: 1999999000000</td>
<td>slow_function took 0.23 sec</td>
</tr>
<tr>
<td>The sum is: 7999998000000</td>
<td>slow_function took 0.41 sec</td>
</tr>
<tr>
<td>The sum is: 31999996000000</td>
<td>slow_function took 0.81 sec</td>
</tr>
<tr>
<td>The sum is: 127999992000000</td>
<td>slow_function took 1.52 sec</td>
</tr>
<tr>
<td>The sum is: 511999984000000</td>
<td>slow_function took 3.12 sec</td>
</tr>
</tbody>
</table>
Built-in @property

- decorator specific for class methods
- allows accessing `x.attribute()` as `x.attribute`, convenient if `attribute` does not take any arguments (also readonly)

```python
rectangle1.py

```class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

# @property
def area(self):
    return self.width * self.height
```

```python
Python shell

```> r = Rectangle(3, 4)
> print(r.area())
12
```

```python
rectangle2.py

```class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

@property
def area(self):
    return self.width * self.height
```

```python
Python shell

```> r = Rectangle(3, 4)
> print(r.area())
12
```
Class decorators

Python
@dec2
@dec1
class A:
    pass

≡

Python
class A:
    pass
A = dec2(dec1(A))
Module **dataclasses** (Since Python 3.7)

- New (and more configurable) alternative to **namedtuple**

```python
from dataclasses import dataclass

@dataclass
# uses a decorator to add methods to the class
class Person:
    name: str  # uses type annotation to define fields
    appeared: int
    height: str = 'unknown height'  # field with default value

person = Person('Donald Duck', 1934, '3 feet')

person

| Person(name='Donald Duck', appeared=1934, height='3 feet')
| 'Donald Duck'

Person('Mickey Mouse', 1928)

| Person(name='Mickey Mouse', appeared=1928, height='unknown height')
```

[docs.python.org/3/library/dataclasses.html#module-dataclasses](https://docs.python.org/3/library/dataclasses.html#module-dataclasses)

Raymond Hettinger - Dataclasses: The code generator to end all code generators - PyCon 2018
@functools.total_ordering (class decorator)

```python
time_it.py
import functools

@functools.total_ordering
class Student:
    def __init__(self, name, student_id):
        self.name = name
        self.id = student_id
    def __eq__(self, other):
        return (self.name == other.name
                and self.id == other.id)
    def __lt__(self, other):
        my_name = ', '.join(reversed(self.name.split()))
        other_name = ', '.join(reversed(other.name.split()))
        return (my_name < other_name
                or (my_name == other_name and self.id < other.id))

donald = Student('Donald Duck', 7)
gladstone = Student('Gladstone Gander', 42)
grandma = Student('Grandma Duck', 1)

Python shell
> donald < grandma | True
> grandma >= gladstone | False
> grandma <= gladstone | True
> donald > gladstone | False
```

Automatically creates <, <=, >, >= if at least one of the functions is implemented and == is implemented
Summary

- `@decorator_name`
- Python decorators are just syntactic sugar
- Adds functionality to a function without having to augment each call to the function or each return statement in the function
- There are decorators for functions, class methods, and classes
- There are many decorators in the Python Standard Library
- Decorators are easy to use
- ...and (slightly) harder to write
Dynamic programming

- memoization
- decorator memoized
- systematic subproblem computation
Binomial coefficient

\[
\binom{n}{k} = \begin{cases} 
1 & \text{if } k = 0 \text{ or } k = n \\
\binom{n-1}{k} + \binom{n-1}{k-1} & \text{otherwise}
\end{cases}
\]

**binomial_recursive.py**

```python
def binomial(n, k):
    if k == 0 or k == n:
        return 1
    return binomial(n - 1, k) + binomial(n - 1, k - 1)
```

**recursion tree for binomial(7, 5)**
Dynamic Programming
≡
Remember solutions already found (memoization)

- Technique sometimes applicable when running time otherwise becomes exponential
- Only applicable if stuff to be remembered is manageable
**Binomial Coefficient**

**Dynamic programming using a dictionary**

```
binomial_dict.py

answers = {}  # answers[(n, k)] = binomial(n, k)

def binomial(n, k):
    if (n, k) not in answers:
        if k == 0 or k == n:
            answer = 1
        else:
            answer = binomial(n - 1, k) + binomial(n - 1, k - 1)
        answers[(n, k)] = answer
    return answers[(n, k)]

Python shell

> binomial(6, 3)
| 20
> answers
```

- Use a dictionary `answers` to store already computed values
- Reuse value stored in dictionary `answers`
Question – What is the order of the size of the dictionary `answers` after calling `binomial(n, k)`?

**binomial_dictionary.py**

```python
answers = {}  # answers[(n, k)] = binomial(n,k)

def binomial(n, k):
    if (n, k) not in answers:
        if k==0 or k==n:
            answer = 1
        else:
            answer = binomial(n-1, k) + binomial(n-1, k-1)
        answers[(n, k)] = answer
    return answers[(n,k)]
```

a) max(n, k)  

b) n + k  

(c) n * k  

d) n^k  

e) k^n  

f) Don’t know
Binomial Coefficient
Dynamic programming using decorator

- Use a decorator (@memoize) that implements the functionality of remembering the results of previous function calls

```python
def memoize(f):
    # answers[args] = f(*args)
    answers = {}
    def wrapper(*args):
        if args not in answers:
            answers[args] = f(*args)
        return answers[args]
    return wrapper

@memoize
def binomial(n, k):
    if k==0 or k==n:
        return 1
    else:
        return binomial(n-1, k) + binomial(n-1, k-1)
```

```python
def trace(f):
    # decorator to trace recursive calls
    indent = 0
    def wrapper(*args):
        nonlocal indent
        spaces = '| ' * indent
        arg_str = ', '.join(map(repr, args))
        print(spaces + f'{f.__name__}({arg_str})')
        indent += 1
        result = f(*args)
        indent -= 1
        print(spaces + f'>{result}')
    return result

def memoize(f):
    @functools.wraps(f)
    def wrapper(*args):
        if args not in answers:
            answers[args] = f(*args)
        return answers[args]
    wrapper.__name__ = f.__name__ + '_memoize'
    return wrapper

@trace
@memoize
def binomial(n, k):
    if k == 0 or k == n:
        return 1
    return binomial(n - 1, k) + binomial(n - 1, k - 1)

print(binomial(5, 2))
```
Dynamic programming using `lru_cache` decorator

The decorator `@lru_cache` in the standard library `functools` supports the same as the decorator `@memoize`.

By default it at most remembers (caches) 128 previous function calls, always evicting Least Recently Used entries from its dictionary.

```python
from functools import lru_cache

@lru_cache(maxsize=None)
def binomial(n, k):
    if k==0 or k==n:
        return 1
    else:
        return binomial(n-1, k) + binomial(n-1, k-1)
```

[docs.python.org/3/library/functools.html#functools.lru_cache](https://docs.python.org/3/library/functools.html#functools.lru_cache)
Subset sum using dynamic programming

- In the subset sum problem (Exercise 13.4) we are given a number \( x \) and a list of numbers \( L \), and want to determine if a subset of \( L \) has sum \( x \)

\[
L = [3, 7, 2, 11, 13, 4, 8] \quad x = 22 = 7 + 11 + 4
\]

- Let \( S(v, k) \) denote if it is possible to achieve value \( v \) with a subset of \( L[:k] \), i.e. \( S(v, k) = \text{True} \) if and only if a subset of the first \( k \) values in \( L \) has sum \( v \)

- \( S(v, k) \) can be computed from the following recurrence:

\[
S(v, k) = \begin{cases} 
\text{True} & \text{if } k = 0 \text{ and } v = 0 \\
\text{False} & \text{if } k = 0 \text{ and } v \neq 0 \\
S(v, k-1) \text{ or } S(v - L[k-1], k-1) & \text{otherwise}
\end{cases}
\]
Subset sum using dynamic programming

```python
subset_sum_dp.py

def subset_sum(x, L):
    @memoize
    def solve(value, k):
        if k == 0:
            return value == 0
        return solve(value, k-1) or solve(value - L[k-1], k-1)
    return solve(x, len(L))

Python shell

> subset_sum(11, [2, 3, 8, 11, -1])
| True
> subset_sum(6, [2, 3, 8, 11, -1])
| False
```
Question – What is a bound on the size order of the memoization table if all values are positive integers?

subset_sum_dp.py

```python
def subset_sum(x, L):
    @memoize
    def solve(value, k):
        if k == 0:
            return value == 0
        return solve(value, k-1) or solve(value - L[k-1], k-1)
    return solve(x, len(L))
```

Python shell

```
> subset_sum(11, [2, 3, 8, 11, -1])
| True
> subset_sum(6, [2, 3, 8, 11, -1])
| False
```

a) `len(L)`
b) `sum(L)`
c) `x`
d) $2^{\text{len}(L)}$
e) `len(L)`
f) `len(L) * sum(L)`
g) Don’t know
Subset sum using dynamic programming

```python
@memoize
def solve(value, k):
    if k == 0:
        if value == 0:
            return []
        else:
            return None
    solution = solve(value, k-1)
    if solution != None:
        return solution
    solution = solve(value - L[k-1], k-1)
    if solution != None:
        return solution + [L[k-1]]
    return None

def subset_sum_solution(x, L):
    return solve(x, len(L))
```

Python shell:

```
> subset_sum_solution(11, [2, 3, 8, 11, -1])
[3, 8]
> subset_sum_solution(6, [2, 3, 8, 11, -1])
None
```
Knapsack problem

- **Given a knapsack** with volume capacity $C$, and set of objects with different volumes and value.

- **Objective**: Find a subset of the objects that fits in the knapsack (sum of volume $\leq$ capacity) and has maximal value.

- **Example**: If $C = 5$ and the volume and weights are given by the table, then the maximal value 15 can be achieved by the 2nd and 3rd object.

- Let $V(c, k)$ denote the maximum value achievable by a subset of the first $k$ objects within capacity $c$.

$$V(c, k) = \begin{cases} 
0 & \text{if } k = 0 \\
V(c, k - 1) & \text{volume}[k - 1] > c \\
\max\{V(c, k - 1), \text{value}[k - 1] + V(c - \text{volume}[k - 1], k - 1)\} & \text{otherwise}
\end{cases}$$
Knapsack – maximum value

```python
knapsack.py
def knapsack_value(volume, value, capacity):
    @memoize
def solve(c, k):
        # solve with capacity c and objects 0..k-1
        if k == 0:
            return 0
        v = solve(c, k - 1)  # try without object k-1
        if volume[k - 1] <= c:
            # try also with object k-1 if space
            v = max(v, value[k - 1] + solve(c - volume[k - 1], k - 1))
        return v
    return solve(capacity, len(volume))

Python shell
>>> volumes = [3, 3, 2, 5]
>>> values = [6, 7, 8, 9]
>>> knapsack_value(volumes, values, 5)
15
```
Knapsack – maximum value and objects

knapsack.py

def knapsack(volume, value, capacity):
    @memoize
    def solve(c, k):
        # solve with capacity c and objects 0..k-1
        if k == 0:
            # no objects to put in knapsack
            return 0, []
        v, solution = solve(c, k-1)
        # try without object k-1
        if volume[k-1] <= c:
            # try also with object k-1 if space
            v2, sol2 = solve(c - volume[k-1], k-1)
            v2 = v2 + value[k-1]
            if v2 > v:
                v = v2
                solution = sol2 + [k-1]
        return v, solution
    return solve(capacity, len(volume))

Python shell

> volumes = [3, 3, 2, 5]
> values = [6, 7, 8, 9]
> knapsack(volumes, values, 5)
| (15, [1, 2])
Knapsack - Table

\[
V(c, k) = \begin{cases} 
0 & \text{if } k = 0 \\
V(c, k - 1) & \text{value}[k-1] > c \\
\max\{V(c, k - 1), \text{value}[k - 1] + V(c - \text{volume}[k - 1], k - 1)\} & \text{otherwise}
\end{cases}
\]

- systematic fill out table
- only need to remember two rows
Knapsack – Systematic table fill out

```python
def knapsack(volume, value, capacity):
    solutions = [(0, [])] * (capacity + 1)
    for obj in range(len(volume)):
        for c in reversed(range(volume[obj], capacity + 1)):
            prev_v, prev_solution = solutions[c - volume[obj]]
            v = value[obj] + prev_v
            if solutions[c][0] < v:
                solutions[c] = v, prev_solution + [obj]
    return solutions[capacity]
```

Python shell

```
> volumes = [3, 3, 2, 5]
> values = [6, 7, 8, 9]
> knapsack(volumes, values, 5)
(15, [1, 2])
```
Summary

- Dynamic programming is a general approach for recursive problems where one tries to avoid recomputing the same expressions repeatedly.

- **Solution 1: Memoization**
  - add dictionary to function to remember previous results
  - decorate with a `@memoize` decorator

- **Solution 2: Systematic table fill out**
  - can need to compute more values than when using memoization
  - can discard results not needed any longer (reduced memory usage)
Visualization and optimization

- Matplotlib
- Jupyter
- scipy.optimize.minimize
Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits.

Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, error charts, scatter plots, etc., with just a few lines of code. For simple plotting the pyplot module provides a MATLAB-like interface, particularly when combined with IPython. For the power user, you have full control of line styles, font properties, axes properties, etc., via an object oriented interface or via a set of functions familiar to MATLAB users.

pip install matplotlib
**Plot**

The `pyplot` module is similar to a MATLAB-like plotting framework.

```python
import matplotlib.pyplot as plt
plt.plot([1, 2, 3], [5, 2, 7], 'bo:');
plt.show()
```

- **Colors**
  - b: Blue
  - g: Green
  - r: Red
  - c: Cyan
  - m: Magenta
  - y: Yellow
  - k: Black
  - w: White

- **Line styles**
  - -: Dashed
  - --: Double dashed
  - -.: Dash-dot
  - -.: Dotted

- **Marker styles**
  - .: Circle
  - : Dot
  - *: Asterisk
  - x: Cross
  - +: Plus
  - v: Triangle down
  - ^: Triangle up
  - <: Triangle left
  - >: Triangle right
  - s: Square
  - p: Pentagon
  - d: Diamond

- **Format string**
  - 'bo:'

- **Figure is first shown when show is called.**

- **Save current view as picture.**
- **Adjust margins.**
- **Zoom rectangle.**
- **Pan and zoom.**
- **Navigate view history.**
- **Reset view.**
Plot – some keyword arguments

```python
import matplotlib.pyplot as plt
X = range(-10, 11)
Y1 = [x ** 2 for x in X]
Y2 = [x ** 3 / 10 + x ** 2 / 2 for x in X]
plt.plot(X, Y1, color='red', label='$x^2$',
         linestyle='-', linewidth=2,
         marker='o', markersize=4,
         markeredgewidth=1,
         markeredgecolor='black',
         markerfacecolor='yellow')
plt.plot(X, Y2, '*', dashes=(2, 0.5, 2, 1.5),
         label=r'$\frac{1}{10}x^3+\frac{1}{2}x^2$')
plt.xlim(-15, 15)
plt.ylim(-75, 125)
plt.title('Some polynomials (degree 2 and 3)')
plt.xlabel('The x-axis')
plt.ylabel('The y-axis')
plt.legend(title='Curves')
plt.show()  # finally show figure
```

matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html
Colors: matplotlib.org/gallery/color/named_colors.html
Scatter (points with individual size and color)

```python
import matplotlib.pyplot as plt
n = 13
X = range(n)
S = [x ** 2 for x in X]
E = [2 ** x for x in X]
plt.scatter(X, [4] * n, s=E, label='s = $2^x$', alpha=.2)
plt.scatter(X, [3] * n, s=S, label='s = $x^2$')
plt.scatter(X, [2] * n, s=X, label='s = $x$')
plt.scatter(X, [1] * n, s=S, c=X, cmap='plasma',
           label='s = $x^2$, c = $x$,
                edgecolors='gray', linewidth=0.5)
plt.colorbar()
plt.ylim(0.5, 5.5)
plt.xlim(0.5, 13.5)
plt.title('A scatter plot')
plt.legend(loc='upper center', frameon=False, ncol=4,
           handletextpad=0)
plt.show()
```

manual placement of legend box (default automatic); remove frame; place legends in 4 columns (default 1); reduce space between marks and label

colorbar (of most recently used colormap)

colormap (predefined)

- color of each point
- size ≈ area of each point
- point boundary width
- point boundary color

transparency

matplotlib.org/api/as_gen/matplotlib.pyplot.scatter.html
matplotlib.org/tutorials/colors/colormaps.html
import matplotlib.pyplot as plt

x = [1, 2, 3]
y = [7, 5, 10]

plt.bar(x, y,
        color='lightblue',  # bar background color
        linewidth=1,       # bar boundary width
        edgecolor='gray',  # bar boundary color
        tick_label=x,      # ticks on x-axis
        width=0.7,         # width, default 0.8
        yerr=0.25,         # Error bar: y length
        xerr=0.5,          # x length
        capsize=3,         # capsize in points
        ecolor='darkblue', # error bar color
        log=True)          # y-axis log scale

plt.bar(x, [v**2 for v in x],
        color='pink',
        linewidth=1,
        edgecolor='gray')

plt.show()}
import matplotlib.pyplot as plt
from random import random
values1 = [random()**2 for _ in range(1000)]
values2 = [random()**3 for _ in range(100)]
bins = [0.0, 0.25, 0.5, 0.75, 1.0]
for i, ht in enumerate(['bar', 'barstacked', 'step', 'stepfilled'], start=1):
    plt.subplot(2, 2, i)  # start new plot
    plt.hist([values1, values2],  # data sets
              bins,  # bucket boundaries
              histtype=ht,  # default ht='bar'
              rwidth=0.7,  # fraction of bucket width
              label=['$x^2$', '$x^3$'],  # labels
              density=True)  # norm. prob. density
    plt.title(ht)  # plot title
    plt.xticks(bins)  # ticks on x-axis
    plt.legend()
plt.suptitle('Histogram')  # figure title
plt.show()
matplotlib-pie.py

```python
import matplotlib.pyplot as plt
plt.title('My Pie')
plt.pie([2, 3, 2, 7], # relative wedge sizes
labels=['A','B','C','D'],
colors=['r', 'b', 'y', 'm'],
explode=(0, 0.1, 0.3, 0), # radius fraction
startangle=5, # angle above horizontal
counterclock=True, # default True
rotatelabels=False, # default False
shadow=True, # default False
textprops=dict( # text properties, dict
color='black', # text color
style='italic'), # text style
wedgeprops=dict( # wedge properties, dict
width=0.8, # width (missing center)
linewidth=1, # wedge boundary width
edgecolor='black'), # boundary color
autopct='%1.1f %%') # percent formatting
plt.show()
```

[matplotlib.org/api/as_gen/matplotlib.pyplot.pie.html](http://matplotlib.org/api/as_gen/matplotlib.pyplot.pie.html)
import matplotlib.pyplot as plt

x = [1, 2, 3, 4]
y1 = [1, 2, 3, 4]
y2 = [2, 3, 1, 4]
y3 = [2, 4, 1, 3]

plt.style.use('dark_background')
for i, base in enumerate(['zero', 'sym', 'wiggle', 'weighted_wiggle'], start=1):
    plt.subplot(4, 1, i)
    plt.stackplot(x, y1, y2, y3,
                  colors=['r', 'g', 'b'],
                  labels=['Red', 'Green', 'Blue'],
                  baseline=base)
    plt.grid(axis='both', # 'x', 'y', or 'both'
             linewidth=0.5, linestyle='-', alpha=0.5)
    plt.legend(title=base, loc='upper left')
    plt.xticks(x) # a tick for each value in x

plt.suptitle('Stackplot')
plt.show()
Subplot

(2 rows, 3 columns)

- Subplots are numbered 1..6 row-by-row, starting top-left
- subplot returns an `axes` to access the plot in the figure

matplotlib.org/api/_as_gen/matplotlib.pyplot.subplot.html
Subplots

```python
import matplotlib.pyplot as plt
from math import pi, sin, cos

times = [2 * pi * t / 1000 for t in range(1001)]

fig, ((ax1, ax2), (ax3, ax4), (ax5, ax6)) = plt.subplots(3, 2, sharex=True, sharey=True)

for i, ax in enumerate([ax1, ax2, ax3, ax4, ax5, ax6], start=1):
    x = [i * sin(i * t) for t in times]
    y = [i * cos(3 * t) for t in times]
    ax.plot(x, y, label=f'$i = {i}$')  # plot to axes
    ax.legend(loc='upper right')  # axes legend

fig.suptitle('subplots', fontsize=16)  # figure title

plt.show()
```

create 6 axes in 3 rows with 2 columns
share the x- and y-axis ranges (automatically applies label_outer to created axes)
returns a pair (figure, axes)

matplotlib.org/api/_as_gen/matplotlib.pyplot.subplots.html
```python
import matplotlib.pyplot as plt
import math

x_min, x_max, n = 0, 2 * math.pi, 20

x = [x_min + (x_max - x_min) * i / n for i in range(n + 1)]
y = [math.sin(v) for v in x]

plt.subplot2grid((5, 5), (0,0), rowspan=3, colspan=3)
plt.fill_between(x, 0.0, y, alpha=0.25, color='r')
plt.plot(x, y, 'r-')
plt.title('Plot A')

plt.subplot2grid((5, 5), (0,3), rowspan=2, colspan=2)
plt.plot(x, y, 'g.')
plt.title('Plot B')

plt.subplot2grid((5, 5), (2,3), rowspan=1, colspan=2)
plt.plot(x, y, 'b--')
plt.title('Plot C')

plt.subplot2grid((5, 5), (3,0), rowspan=2, colspan=5)
plt.plot(x, y, 'mx:')
plt.title('Plot D')

plt.tight_layout()  # adjust padding
plt.show()
```
There are many ways to make the x- and/or y-axis logarithmic with pyplot.
import matplotlib.pyplot as plt
from math import pi, sin, cos
n = 1000
points = [(cos(2 * pi * i / n),
           sin(4 * pi * i / n)) for i in range(n)]
x, y = zip(*points)
plt.plot(x, y, 'k-', linewidth=5)
plt.savefig('butterfly.png')  # save plot as PNG
plt.savefig('butterfly-grey.png',
            dpi=100,                  # dots per inch
            bbox_inches='tight',     # crop to bounding box
            pad_inches=0.1,          # space around figure
            facecolor='lightgrey',   # background color
            format='png')            # optional if file extension
plt.savefig('butterfly.pdf')  # save plot as PDF
plt.show()                      # interactive viewer
plot returns “Line2D” objects representing the plotted data. “Line2D” objects can be updated using set_data.

To make an animation you need to repeatedly update the “line2D” objects.

FuncAnimation repeatedly calls func in regular intervals interval, each time with the next value from frames (if frames is None, then the frame values provided to func will be the infinite sequence 0,1,2,3,...)

matplotlib.org/api/_as_gen/matplotlib.animation.FuncAnimation.html
The Jupyter Notebook
The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.
Prime Number Theorem

\( \pi(n) \) is the number of prime numbers \( \leq n \). The Prime Number Theorem states that \( \pi(n) \approx \frac{n}{\log n} \).

In the following we consider all primes \( \leq 1,000,000 \). First we compute a set 'composite' of all composite numbers in the range \( 2 \leq n \).

In [1]:
composite = [p for f in range(2, n + 1) for p in range(f, n + 1, f)]

We next compute select all the prime numbers in the range \( 2 \leq n \), i.e., the non-composite numbers.

In [2]:
primes = [p for p in range(2, n + 1) if p not in composite]

In [3]:
primes[:10]
Out[3]:
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]

In [4]:
import matplotlib.pyplot as plt
import math

X = range(2, n + 1, 25000)
Y = [len((p for p in primes if p \leq m)) for m in X] if slow
plt.plot(X, Y, 'g')
plt.plot(X, [x / math.log(x) for x, y in zip(X, Y)], '-r')
plt.show()
**Prime Number Theorem**

\( \pi(n) = \text{the number of prime numbers } \leq n \). The Prime Number Theorem states that \( \pi(n) \approx \frac{n}{\log n} \).

In the following we consider all primes \( \leq 1,000,000 \). First we compute a set 'composite' of all composite numbers in the range \( 1 \leq n \).

```python
In [1]: n = 1_000_000
   composite = [p for f in range(2, n+1) for p in range(f, n+1, f)]
```

We next compute select all the prime numbers in the range \( 2 \leq n \), i.e. the non-composite numbers.

```python
In [2]: primes = [p for p in range(2, n+1) if p not in composite]
In [3]: primes[10]
Out[3]: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

```python
In [4]: from matplotlib.pyplot import plt
   import math

   X = range(2, n+1, 25000)
   Y = [len([p for p in primes if p == x]) for x in X] # slow
   plt.plot(X, Y, 'g')
   plt.plot(X, [x / math.log(x) for x, y in zip(X, Y)], 'r-')
   plt.show()
```

![Graph](image)
Jupyter - installing

- Open a windows shell and run: `pip install jupyter`
Jupyter – launching the jupyter server

- Open a windows shell and run: `jupyter notebook`

- If this does not work, then try `python -m notebook`
Try:
Help > User Interface Tour
Help > Markdown

# Title of markdown cell

1. first item
2. second item

Here \textit{comes} \textbf{some} math $x^2 \cdot \left(\int_1^n x \, dx\right)$

Title of markdown cell

1. first item
2. second item

Here \textit{comes} \textbf{some} math $x^2 \cdot \left(\int_1^n x \, dx\right)$
Command Mode

- Used to navigate between cells
- Current cell is marked with blue bar
- Keyboard shortcuts

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>show keyboard shortcuts</td>
</tr>
<tr>
<td>enter</td>
<td>enter Edit Mode on current cell</td>
</tr>
<tr>
<td>shift-enter</td>
<td>run cell + select below</td>
</tr>
<tr>
<td>ctrl-enter</td>
<td>run selected cells</td>
</tr>
<tr>
<td>alt-enter</td>
<td>run cell and insert below</td>
</tr>
<tr>
<td>Y M R</td>
<td>change cell type (code, markdown, raw text)</td>
</tr>
<tr>
<td>1 2 3 4 5 6</td>
<td>change heading level</td>
</tr>
<tr>
<td>ctrl-A</td>
<td>select all cells</td>
</tr>
<tr>
<td>down up</td>
<td>move to next/previous cell</td>
</tr>
<tr>
<td>space shift-space</td>
<td>scroll down/up</td>
</tr>
<tr>
<td>shift-up shift-down</td>
<td>extend selected cells</td>
</tr>
<tr>
<td>A B</td>
<td>insert cell above/below</td>
</tr>
<tr>
<td>X C V</td>
<td>cut, copy, paste below/above, undo, delete cells</td>
</tr>
<tr>
<td>shift-L</td>
<td>toggle line numbers in cells</td>
</tr>
<tr>
<td>shift-M</td>
<td>merge selected cells (or with cell below)</td>
</tr>
<tr>
<td>O</td>
<td>toggle output of selected cells</td>
</tr>
<tr>
<td>shift-O</td>
<td>toggle scrollbar on selected cells (long output)</td>
</tr>
</tbody>
</table>
Edit Mode

- Used to edit current cell
- Current cell is marked with green bar
- Keyboard shortcuts

<table>
<thead>
<tr>
<th>Key Combination</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>esc</code></td>
<td>Enter Command Mode</td>
</tr>
<tr>
<td><code>shift-enter</code></td>
<td>Run cell + select below</td>
</tr>
<tr>
<td><code>ctrl-enter</code></td>
<td>Run selected cells</td>
</tr>
<tr>
<td><code>alt-enter</code></td>
<td>Run cell and insert below</td>
</tr>
<tr>
<td><code>ctrl-shift-</code></td>
<td>Split cell at cursor</td>
</tr>
<tr>
<td><code>ctrl-shift-f</code></td>
<td>Command palette</td>
</tr>
<tr>
<td><code>tab</code></td>
<td>Indent or code completion</td>
</tr>
<tr>
<td><code>shift-tab</code></td>
<td>Show docstring</td>
</tr>
<tr>
<td><code>ctrl-a - x - c - v - z - y</code></td>
<td>Select all, cut, copy, paste, undo, redo</td>
</tr>
<tr>
<td><code>ctrl-d</code></td>
<td>Delete line</td>
</tr>
</tbody>
</table>
Evaluating cells

- To evaluate cell: ctrl-enter, alt-enter, shift-enter
- Output from program shown below cell
- Result of last evaluated line
- Order of code cells evaluated
  - Note "x ** 2" computed after "x = 4"
- [*] are cells being evaluated / waiting
- [ ] not yet evaluated
- Recompute all cells top-down
  - or Kernel > Restart & Run all
### Magic lines

- Jupyter code cells support **magic commands** (actually it is IPython)
- `%` is a **line magic**
- `%%` is a **cell magic**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%smagic</td>
<td>list magic commands</td>
</tr>
<tr>
<td>%quickref</td>
<td>quick reference sheet to IPython</td>
</tr>
<tr>
<td>%pwd</td>
<td>print working directory (current folder)</td>
</tr>
<tr>
<td>%cd directory</td>
<td>change directory (absolute or relative)</td>
</tr>
<tr>
<td>%ls</td>
<td>list content of current directory</td>
</tr>
<tr>
<td>%pip or %conda</td>
<td>run pip or conda from jupyter</td>
</tr>
<tr>
<td>%load script</td>
<td>insert external script into cell</td>
</tr>
<tr>
<td>%run program</td>
<td>run external program and show output</td>
</tr>
<tr>
<td>%automagic</td>
<td>toggle if %-prefix is required</td>
</tr>
<tr>
<td>%matplotlib inline</td>
<td>no zoom &amp; resize, allows multiple plots</td>
</tr>
<tr>
<td>%matplotlib notebook</td>
<td>a single plot can be zoomed &amp; resized</td>
</tr>
<tr>
<td>%writefile file</td>
<td>write content of cell to a file</td>
</tr>
<tr>
<td>%time</td>
<td>measure time for cell execution</td>
</tr>
<tr>
<td>%timeit expression</td>
<td>time for simple expression</td>
</tr>
</tbody>
</table>
Jupyter and matplotlib

- `%matplotlib inline`  
  pyplot figures are shown *without* interactive zoom and pan (default)
- Consider changing default figure size  
  `plt.rcParams['figure.figsize']`
- Start each figure with `plt.figure`
- Final call to `show` can be omitted
Jupyter and matplotlib

- `%matplotlib notebook
  pyplot figures are shown with interactive zoom and pan
- Start each figure with `plt.figure`
  (also allows setting figure size)
- Final call to `show` can be omitted
- Widespread tool used for data science applications
- Documentation, code for data analysis, and resulting visualizations are stored in one common format
- Easy to update visualizations
- Works with about 100 different programming languages (not only Python 3), many special features, ....
- Easy to share data analysis

Many online tutorials and examples are available

https://www.youtube.com/results?search_query=jupyter+python
scipy.optimize.minimize

- Find point $p$ minimizing function $f$
- Supports 13 algorithms – but no guarantee that result is correct
- Knowledge about optimization will help you know what optimization algorithm to select and what parameters to provide for better results
- **WARNING**
  Many solvers return the wrong value when used as a black box

docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html
```python
from math import sin
import matplotlib.pyplot as plt
from scipy.optimize import minimize

trace = []  # remember calls to f

def f(x):
    value = x ** 2 + 10 * sin(x)
    trace.append((x, value))
    return value

X = [-8 + 18 * i / 9999 for i in range(1000)]
Y = [f(x) for x in X]

plt.style.use('dark_background')
plt.plot(X, Y, 'w-')

for start, color in [(8, 'red'), (-6, 'yellow')]:
    trace = []
    solution = minimize(f, [start], method='nelder-mead')
    x, y = solution.x[0], solution.fun
    plt.plot(*zip(*trace), 'o', c=color)
    plt.plot(x, -23, f'x: {x:.3f}', c=color, ha='center')

plt.xticks(range(-5, 15, 5))
plt.yticks(range(-25, 100, 25))
plt.minorticks_on()
plt.legend()
plt.show()

print(solution)
```

```
Example: Minimum enclosing circle

- Find \( c \) such that \( r = \max_p |p - c| \) is minimized

- A solution is characterized by either
  1) three points on circle, where the triangle contains the circle center
  2) two opposite points on diagonal

- Try a standard numeric minimization solver

- Caution: Computation involves \( \max \) and \( \sqrt{x} \), which can be hard for numeric optimization solvers
Some basic differences

- "end" closes a MATLAB block
- ";" at end of command avoids command output
- a(i) instead of a[i]
- 1st element of a list a(1)
- a(i:j) includes both a(i) and a(j)

like R, Mathematica, Julia, AWK, Smalltalk, ...
% Minimum enclosing circle of a point set
% fminsearch uses the Nelder-Mead algorithm

global x y
x = [1.0, 3.0, 2.5, 4.0, 5.0, 6.0, 5.0];
y = [3.0, 1.0, 3.0, 6.0, 7.0, 7.0, 2.0];
c = fminsearch(@(x) max_distance(x), [0,0]);
plot(x, y, "o");
viscircles(c, max_distance(c));

function dist = max_distance(p)
    global x y
    dist = 0.0;
    for i=1:length(x)
        dist = max(dist, pdist([p; x(i), y(i)],
                                'euclidean'));
    end
end
Minimum enclosing circle in MATLAB (trace)

enclosing_circle_trace.m

```matlab
global x y trace_x trace_y

x = [1.0, 3.0, 2.5, 4.0, 5.0, 6.0, 5.0];
y = [3.0, 1.0, 3.0, 6.0, 7.0, 7.0, 2.0];
trace_x = [];
trace_y = [];

c = fminsearch(@(x) max_distance(x), [0,0]);
hold on
plot(x, y, "o", 'color', 'b', 'MarkerFaceColor', 'b');
plot(trace_x, trace_y, "*-", "color", "g");
plot(c(1),c(2), "o", 'color', 'r', 'MarkerFaceColor', 'r');
viscircles(c, max_distance(c),"color","red");

function dist = max_distance(p)
    global x y trace_x trace_y
    trace_x = [trace_x, p(1)];
    trace_y = [trace_y, p(2)];

    dist = 0.0;
    for i=1:length(x)
        dist = max(dist, pdist([p; x(i), y(i)], 'euclidean'));
    end
end
```
Minimum enclosing circle in Python

```python
closing_circle.py

from math import sqrt
from scipy.optimize import minimize
import matplotlib.pyplot as plt

x = [1.0, 3.0, 2.5, 4.0, 5.0, 6.0, 5.0]
y = [3.0, 1.0, 3.0, 6.0, 7.0, 7.0, 2.0]

def dist(p, q):
    return sqrt((p[0] - q[0])**2 + (p[1] - q[1])**2)

def max_distance(c):
    return max(dist(p, c) for p in zip(x, y))

c = minimize(max_distance, [0.0, 0.0],
              method="nelder-mead").x

ax = plt.gca()
ax.set_xlim((0, 8))
ax.set_ylim((0, 8))
ax.set_aspect("equal")
plt.plot(x, y, "g.")
ax.add_artist(plt.Circle(c, max_distance(c),
                        color="r", fill=False))
plt.show()
```
from math import sqrt
from scipy.optimize import minimize
import matplotlib.pyplot as plt

x = [1.0, 3.0, 2.5, 4.0, 5.0, 6.0, 5.0]
y = [3.0, 1.0, 3.0, 6.0, 7.0, 7.0, 2.0]
trace = []

def dist(p, q):
    return sqrt((p[0]-q[0])**2 + (p[1]-q[1])**2)

def max_distance(c):
    trace.append(c)
    return max(dist(p, c) for p in zip(x, y))

c = minimize(max_distance, [0.0, 0.0],
              method="nelder-mead").x

ax = plt.gca()
ax.set_xlim((0, 8))
ax.set_ylim((0, 8))
ax.set_aspect("equal")
plt.plot(x, y, "g.")
plt.plot(*zip(*trace), "b.-")
ax.add_artist(plt.Circle(c, max_distance(c),
                        color="r", fill=False))
plt.show()
Minimum enclosing circle – search space

Maximum distance to an input point
```python
from math import sqrt
from scipy.optimize import minimize
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

points = [(1.0, 3.0), (3.0, 1.0), (2.5, 3.0),
          (4.0, 6.0), (5.0, 7.0), (6.0, 7.0), (5.0, 2.0)]

# Minimum enclosing circle solver
trace = []

def distance(p, q):
    return sqrt((p[0] - q[0])**2 + (p[1] - q[1])**2)

def distance_max(q):
    dist = max(distance(p, q) for p in points)
    trace.append((q, dist))
    return dist

solution = minimize(distance_max, [0.0, 0.0],
                     method='nelder-mead')

center = solution.x
radius = solution.fun

# unzip point coordinates
points_x, points_y = zip(*points)
trace_x, trace_y, trace_z = zip(*trace)

# Bounding box [x_min, x_max] x [y_min, y_max]
xs, ys = points_x + trace_x, points_y + trace_y
x_min, x_max = min(xs), max(xs)
y_min, y_max = min(ys), max(ys)

# enforce aspect ratio
x_max = max(x_max, x_min + y_max - y_min)
y_max = max(y_max, y_min + x_max - x_min)

# Minimum enclosing circle - 3D surface plot
X, Y = np.meshgrid(np.linspace(x_min, x_max, 100),
                   np.linspace(y_min, y_max, 100))
Z = np.zeros(X.shape)
for px, py in points:
    Z = np.maximum(Z, (X - px)**2 + (Y - py)**2)
Z = np.sqrt(Z)

ax = plt.subplot(1, 2, 1, projection='3d')
ax.plot_surface(X, Y, Z, cmap='plasma', alpha=0.7)
ax.plot(trace_x, trace_y, trace_z, '.', c='darkblue')
ax.scatter(*center, radius, 'o', c='red')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('max distance')
ax.set_title('plot_surface')

# Minimum enclosing circle - contour plot
plt.subplot(1, 2, 2)
plt.title('pyplot.contour')
plt.plot(trace_x, trace_y, '.', color='darkblue')
plt.plot(points_x, points_y, 'o', color='darkgreen')
plt.plot(*center, 'o', c='red')
qcs = plt.contour(X, Y, Z, levels=30, cmap='plasma')
plt.clabel(qcs, inline=1, fontsize=8, fmt='%.1f')
plt.suptitle('Maximum distance to an input point')
plt.tight_layout()
plt.show()```
scipy.minimize  $f(c) = \max_{p} |p - c|$
scipy.minimize \( f(c) = \max_p |p - c|^2 \) avoids \( \sqrt{\text{improved}} \)
Multi-dimensional data

- NumPy
- matrix multiplication, @
- numpy.linalg.solve, numpy.polyfit
NumPy is a Python package for dealing with multi-dimensional data.
Guttag uses `pylab` in the examples, but...

"pylab is a convenience module that bulk imports `matplotlib.pyplot` (for plotting) and `numpy` (for mathematics and working with arrays) in a single name space. Although many examples use `pylab`, it is no longer recommended."

[matplotlib.org/faq/usage_faq.html](https://matplotlib.org/faq/usage_faq.html)
NumPy arrays (example)

**Python shell**

```python
> range(0, 1, .3)
  TypeError: 'float' object cannot be interpreted as an integer
> [1 + i / 4 for i in range(5)]
  [1.0, 1.25, 1.5, 1.75, 2.0]
```

Python only supports ranges of int

generate 5 uniform values in range [1,2]

```python
> import numpy as np
> np.arange(0, 1, 0.3)
  array([0. , 0.3, 0.6, 0.9])
> type(np.arange(0, 1, 0.3))
  <class 'numpy.ndarray'>
> help(numpy.ndarray)
  +2000 lines of text
> np.linspace(1, 2, 5)
  array([1. , 1.25, 1.5 , 1.75, 2. ])
```

numpy can generate ranges with float

returns a ”NumPy array” (not a list)

“arange” ≡ ”array range” and generates the array explicitly

generate n uniformly spaced values
Plotting a function (example)

```
sin.py
import matplotlib.pyplot as plt
import math
n = 25
x = [2*math.pi * i / (n-1) for i in range(n)]
y = [math.sin(v) for v in x]
plt.plot(x, y, '.')
plt.show()
```

```
sin_numpy.py
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 2 * np.pi, 25)
y = np.sin(x)
plt.plot(x, y, '.')
plt.show()
```

- `np.sin` applies the `sin` function to each element of `x`
- `pyplot` accepts NumPy arrays
- `math.pi == np.pi ≈ \frac{22}{7}`
A circle

circle.py

import matplotlib.pyplot as plt
import numpy as np
a = np.linspace(0, 2 * np.pi, 100)
x = np.sin(a)
y = np.cos(a)
plt.plot(x, y, 'r-')
plt.plot(x[0], y[0], 'bo')
plt.show()
Two half circles

```python
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-1, 1, 100)
plt.plot(x, np.sqrt(1 - x**2), 'r-')
plt.plot(x, -np.sqrt(1 - x**2), 'b-')
plt.show()
```

- $x^2 + y^2 = 1$
  \[\Rightarrow\]
  \[y = \sqrt{1 - x^2}\]

- $x$ is a NumPy array
- ** Semi-NumPy method `**___**pow**___**` squaring each element in $x$
- ** Binary NumPy method `**___**rsub**___**` that for each element $e$ in $x$ computes $1 - e$
- `np.sqrt` NumPy method computing the square root of each element in $x$
- ** Unary NumPy method `**___**neg**___**` that negates each element in $x$
Creating one-dimensional NumPy arrays

Python shell

```python
> np.array([1, 2, 3])
array([1, 2, 3])
> np.array((1, 2, 3))
array([1, 2, 3])
> np.array(range(1, 4))
array([1, 2, 3])
> np.arange(1., 4., 1.)
array([1., 2., 3.])
> np.linspace(1, 3, 3)
array([1., 2., 3.])
> np.zeros(3)
array([0., 0., 0.])
> np.ones(3)
array([1., 1., 1.])
> np.full(3, 7)
array([7, 7, 7])
> np.random.random(3)
array([0.73761651, 0.60607355, 0.3614118])
```

```python
> np.array([1, 2, 3]).dtype  # type of all values
dtype('int32')
> np.arange(3, dtype='float')
array([0., 1., 2.])
> np.arange(3, dtype='int16')  # 16 bit integers
array([0, 1, 2], dtype=int16)
> np.arange(3, dtype='int32')  # 32 bit integers
array([0, 1, 2])
> 1000 ** np.arange(5)
array([1, 1000, 1000000, 1000000000, -727379968], dtype=int32)  # OOPS.. overflow
> 1000 ** np.arange(5, dtype='O')
array([1, 1000, 1000000, 1000000000, 1000000000000], dtype=object)  # Python integer
> np.arange(3, dtype='complex')
array([0.+0.j, 1.+0.j, 2.+0.j])
```

Elements of a NumPy array are not arbitrary precision integers by default – you can select between +25 number representations.

[docs.scipy.org/doc/numpy-1.13.0/user/basics.types.html](http://docs.scipy.org/doc/numpy-1.13.0/user/basics.types.html)
Mantissa size in various numpy floats

```python
for data_type in ['half', 'float', 'single', 'double', 'longdouble', 'float32', 'float64']:
    x = np.array([1], dtype=data_type)
    for i in range(100):
        if x == x + (x / 2) ** i:
            break
    print(data_type, i - 1, 'bits mantissa')
    # platform independent
```

<table>
<thead>
<tr>
<th>data_type</th>
<th>Mantissa size</th>
</tr>
</thead>
<tbody>
<tr>
<td>half</td>
<td>10 bits mantissa</td>
</tr>
<tr>
<td>float</td>
<td>52 bits mantissa</td>
</tr>
<tr>
<td>single</td>
<td>23 bits mantissa</td>
</tr>
<tr>
<td>double</td>
<td>52 bits mantissa</td>
</tr>
<tr>
<td>longdouble</td>
<td>52 bits mantissa</td>
</tr>
<tr>
<td>float32</td>
<td>23 bits mantissa</td>
</tr>
<tr>
<td>float64</td>
<td>52 bits mantissa</td>
</tr>
</tbody>
</table>

docs.scipy.org/doc/numpy-1.13.0/user/basics.types.html
Creating multi-dimensional NumPy arrays

```python
Python shell

```
```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
View vs Copy

- Numpy is optimized to handle big multi-dimensional arrays
- To avoid copying data, views allows one to look at the underlying data in different ways (data can be shared by multiple views)
- reshape, ravel and slicing are examples creating views
- flatten and ravel both turn multiple dimensional arrays into one dimensional arrays but flatten creates an explicit copy whereas ravel creates a space efficient view
NumPy operations

Python shell

```python
> x = numpy.arange(3)
> x
| array([0, 1, 2])
> x + x  # elementwise addition
| array([0, 2, 4])
> 1 + x  # add integer to each element
| array([1, 2, 3])
> x * x  # elementwise multiplication
| array([0, 1, 4])
> np.dot(x, x)  # dot product
| 5
> np.cross([1, 2, 3], [3, 2, 1])  # cross product
| array([-4,  8, -4])
```

```python
> a = np.arange(6).reshape(2,3)
> a
| array([[0, 1, 2],
|        [3, 4, 5]])
> a.T  # matrix transposition, view
| array([[0, 3],
|        [1, 4],
|        [2, 5]])
> a @ a.T  # matrix multiplication
| array([[ 5, 14],
|        [14, 50]])
> a += 1
> a
| array([[1, 2, 3],
|        [4, 5, 6]])
```

PEP 465 - A dedicated infix operator for matrix multiplication
Universal functions (apply to each entry)

Python shell

```python
x = np.array([[1, 2], [3, 4]])
np.sin(x)  # also: cos, exp, sqrt, log, ceil, floor, abs
# array([[ 0.84147098,  0.90929743],
#         [ 0.14112001, -0.7568025 ]])
np.sign(np.sin(x))
# array([[ 1.,  1.],
#        [ 1., -1.]])
np.mod(np.arange(10), 3)  # same as: np.arange(10) % 3
# array([0, 1, 2, 0, 1, 2, 0, 1, 2, 0], dtype=int32)
```
Axis

Python shell

> x = np.arange(1, 7).reshape(2, 3)
> x
| array([[1, 2, 3],
|        [4, 5, 6]])
> x.sum()  # = x.sum(axis=(0, 1))
| 21
> x.sum(axis=0)
| array([5, 7, 9])
> x.sum(axis=1)
| array([6, 15])
> x.min()  # = x.min(axis=(0, 1))
| 1

Python shell

> x.min(axis=0)
| array([1, 2, 3])
> x.min(axis=1)
| array([1, 4])
> x.cumsum()
| array([ 1,  3,  6, 10, 15, 21], dtype=int32)
> x.cumsum(axis=0)
| array([[1, 3, 6, 10, 15, 21]], dtype=int32)
> x.cumsum(axis=1)
| array([[ 1,  3,  6],
|        [ 4,  9, 15]], dtype=int32)
Slicing

Python shell

```python
> x = numpy.arange(20).reshape(4,5)
> x
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
> x[2, 3]  # = x[(2, 3)]
13
> x[1:4:2, 1:4:1]  # rows 1 and 3, and columns 2 and 3, view
array([[ 7,  8],
       [17, 18]])
> x[:,3]
array([ 3,  8, 13, 18])
> x[... ,3]  # ... is placeholder for ':' for all missing dimensions
array([ 3,  8, 13, 18])
> type(...)  
<class 'ellipsis'>
```
Broadcasting (stretching arrays to get same size)

- Numpy tries to apply **broadcasting**, if array shapes do not match, i.e. adds missing leading dimensions and repeats a dimension with only one element:
  
  \[
  \begin{bmatrix}
  1 \\
  2
  \end{bmatrix} + \begin{bmatrix}
  10 \\
  20
  \end{bmatrix} \quad \text{column + row vector}
  \equiv \begin{bmatrix}
  1 \\
  2
  \end{bmatrix} + \begin{bmatrix}
  10, 20
  \end{bmatrix} \quad \text{both ndim = 2}
  \equiv \begin{bmatrix}
  1, 1 \\
  2, 2
  \end{bmatrix} + \begin{bmatrix}
  10, 20
  \end{bmatrix}
  \equiv \begin{bmatrix}
  1, 1 \\
  2, 2
  \end{bmatrix} + \begin{bmatrix}
  10, 20 \\
  10, 20
  \end{bmatrix}
  \equiv \begin{bmatrix}
  11, 21 \\
  12, 22
  \end{bmatrix}
  \]

- To prevent unexpected broadcasting, add an assertion to your program:

  ```python
  assert x.shape == y.shape
  ```

  [docs.scipy.org/doc/numpy/user/basics.broadcasting.html](http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html)
Masking

```python
x = np.arange(1, 11).reshape(2, 5)
x
| array([[ 1,  2,  3,  4,  5],
        [ 6,  7,  8,  9, 10]])

x % 3
| array([[1, 2, 0, 1, 2],
        [0, 1, 2, 0, 1]], dtype=int32)

x % 3 == 0
| array([[False, False,  True, False, False],
        [ True, False, False,  True, False]])

x[x % 3 == 0] # use Boolean matrix to select entries
| array([3, 6, 9])

x[:, x.sum(axis=0) % 3 == 0] # columns with sum divisible by 3
| array([[ 2,  5],
        [ 7, 10]])
```
Numpy is fast... but be aware of dtype

```python
sum([x**2 for x in range(1000000)])
| 333332833333500000
> (np.arange(1000000)**2).sum()
| 584144992 # wrong since overflow when default dtype='int32'
> (np.arange(1000000, dtype='int64')**2).sum()
| 333332833333500000 # 64 bit integers do not overflow

import timeit
from timeit
> timeit('sum([x**2 for x in range(1000000)])', number=1)
| 0.5614346340007614
> timeit('np.arange(1000000)**2).sum()', setup='import numpy as np', number=1)
| 0.014362967000124627 # ridiculous fast but also wrong result...
> timeit('np.arange(1000000, dtype="int64")**2).sum()',
setup='import numpy as np', number=1)
| 0.048017077999247704 # fast and correct

np.iinfo(np.int32).min
| -2147483648
> np.iinfo(np.int32).max
| 2147483647
```
Linear algebra

Python shell

```python
> x = np.arange(1, 5, dtype=float).reshape(2, 2)
> x
| array([[1., 2.],
|        [3., 4.]]),
> x.T  # matrix transpose
| array([[1., 3.],
|        [2., 4.]]),
> np.linalg.det(x)  # matrix determinant
| -2.0000000000000004
> np.linalg.inv(x)  # matrix inverse
| array([[-2. ,  1. ],
|        [ 1.5, -0.5]]),
> np.linalg.eig(x)  # eigenvalues and eigenvectors
| (array([-0.37228132,  5.37228132]),
| array([[-0.82456484, -0.41597356],
|        [0.56576746, -0.90937671])))
> y = np.array([[5.], [7.]])
> np.linalg.solve(x, y)  # solve linear matrix equations
| array([[-3.],  # z1
|        [4.]]),  # z2
```

It is no longer recommended to use this class, even for linear algebra. Instead use regular arrays. The class may be removed in the future.
numpy.polyfit

- Given \( n \) points with \((x_0, y_0), \ldots, (x_{n-1}, y_{n-1})\)
- Find polynomial \( p \) of degree \( d \) that minimizes
  \[
  \sum_{i=0}^{n-1} (y_i - p(x_i))^2
  \]
- know as least squares fit / linear regression / polynomial regression

docs.scipy.org/doc/numpy/reference/generated/numpy.polyfit.html
```python
import matplotlib.pyplot as plt
import numpy as np

x = [0, 2, 3, 5, 6, 7, 8]
y = [-2, 4, 3, 2, 4, 9, 12]
coefficients = np.polyfit(x, y, 3)

fx = np.linspace(-1, 9, 100)
fy = np.polyval(coefficients, fx)

plt.plot(fx, fy, '-')
plt.plot(x, y, 'ro')
plt.show()
```
Least squares polynomial fit

- degree 1
- degree 2
- degree 5
- degree 10
- average
import matplotlib.pyplot as plt
import numpy as np

x = 3 * np.random.random(25)
noise = np.random.random(x.size) ** 2
y = 5 * x ** 2 - 12 * x + 7 + 5 * noise

for degree in [1, 2, 5, 10]:
    coefficients = np.polyfit(x, y, degree)
    fx = np.linspace(-1, 4, 100)
    fy = np.polyval(coefficients, fx)
    plt.plot(fx, fy, '-', label="degree %s" % degree)

avg = np.average(y)
plt.plot(x, y, 'ro')
plt.plot([-1, 4], [avg, avg], 'k-', label="average")
plt.ylim(-3, 15)
plt.title('Least squares polynomial fit')
plt.legend()
plt.show()
Animating bouncing balls

- matplotlib figures can be animated using `matplotlib.animation.FuncAnimation` that as arguments take the figure to me updated/redrawn, a function to call for each update, and an interval in milliseconds between updates

```python
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from numpy import zeros, maximum, minimum
from numpy.random import random

g = 0.01
N = 10

x, y = 10.0 * random(N), 1.0 + 9.0 * random(N)
dx, dy = random(N) / 5, zeros(N)

fig = plt.figure()
plt.xlim(0, 10)
plt.ylim(0, 10)

balls, = plt.plot(x, y, 'o')  # returns Line2D obj

def move(frame):
    global x, y, dx, dy
    x += dx
    bounce = (x > 10.0) | (x < 0.0)  # numpy mask
    dx[bounce] = -dx[bounce]
    x = minimum(10.0, maximum(0.0, x))

    y += dy
    bounce = y < 0.0  # numpy mask
    y[bounce] = dy[bounce]
    dy[bounce] = -dy[bounce]
    dy -= g

    balls.set_data(x, y)  # update positions

ani = FuncAnimation(fig, move, interval=25)
plt.show()
```
```python
import numpy as np
import matplotlib.pyplot as plt

def mandelbrot(h, w, maxit=20):
    """Returns an image of the Mandelbrot fractal of size (h, w).""
    x = np.linspace(-2.0, 0.8, w).reshape(1, w)  # row vector
    y = np.linspace(-1.4, 1.4, h).reshape(h, 1)   # column vector
    c = x + y * 1j  # broadcast & complex
    z = c
    divtime = np.full(z.shape, maxit, dtype=int)  # all values = maxit
    for i in range(maxit):
        z = z * z + c  # elementwise
        diverge = z * np.conj(z) > 4  # who is diverging
        div_now = diverge & (divtime == maxit)  # who is diverging now
        divtime[div_now] = i  # note when
        z[diverge] = 2  # limit divergence  # (avoids overflows)
    return divtime

plt.imshow(mandelbrot(400, 400))
plt.show()
```
code from docs.scipy.org/doc/numpy/user/quickstart.html
Linear programming

- Example Numpy: PageRank
- scipy.optimize.linprog
- Example linear programming: Maximum flow
PageRank
PageRank - A NumPy / Jupyter / matplotlib example

- Google's original search engine ranked webpages using **PageRank**
- View the internet as a graph where **nodes** correspond to webpages and **directed edges** to links from one webpage to another webpage
- Google’s PageRank algorithm was described in ([ilpubs.stanford.edu:8090/361/](ilpubs.stanford.edu:8090/361/), 1998)

**The Anatomy of a Large-Scale Hypertextual Web Search Engine**

Sergey Brin and Lawrence Page

*Computer Science Department, Stanford University, Stanford, CA 94305, USA*

sergey@cs.stanford.edu and page@cs.stanford.edu
Five different ways to compute PageRank probabilities

1) Simulate random process manually by rolling dices
2) Simulate random process in Python
3) Computing probabilities using matrix multiplication
4) Repeated matrix squaring
5) Eigenvector for $\lambda = 1$
Random surfer model (simplified)

The PageRank of a node (web page) is the fraction of the time one visits a node by performing an infinite random traversal of the graph starting at node 1, and in each step:

- with probability $1/6$ jumps to a random page (probability $1/6$ for each node)
- with probability $5/6$ follows an outgoing edge to an adjacent node (selected uniformly)

The above can be simulated by using a dice: Roll a dice. If it shows 6, jump to a random page by rolling the dice again to figure out which node to jump to. If the dice shows 1-5, follow an outgoing edge - if two outgoing edges roll the dice again and go to the lower number neighbor if it is odd.
import numpy as np

# Adjacency matrix of the directed graph in the figure
# (note that the rows/columns are 0-indexed, whereas in the figure the nodes are 1-indexed)

G = np.array([[0, 1, 0, 0, 0, 0],
              [0, 0, 0, 1, 0, 0],
              [1, 1, 0, 0, 0, 0],
              [0, 1, 0, 0, 1, 0],
              [0, 1, 0, 0, 0, 1],
              [0, 1, 0, 0, 0, 0]])

n = G.shape[0]  # number of rows in G
degree = np.sum(G, axis=1, keepdims=True)  # column vector with row sums = out-degrees

# The below code handles sinks, i.e. nodes with outdegree zero (no effect on the graph above)

G = G + (degree == 0)  # add edges from sinks to all nodes
degree = np.sum(G, axis=1, keepdims=True)
Simulate random walk (random surfer model)

```python
from random import randint

STEPS = 1000000

# adjacency_list[i] is a list of all j where (i, j) is an edge of the graph.
adjacency_list = [[j for j, e in enumerate(row) if e] for row in G]

count = np.zeros(n)  # histogram over number of node visits
state = 0  # start at node with index 0

for _ in range(STEPS):
    count[state] += 1  # increment count for state
    if randint(1, 6) == 6:  # original paper uses 15% instead of 1/6
        state = randint(0, 5)
    else:
        state = adjacency_list[state][randint(0, degree[state] - 1)]

print(adjacency_list, count / STEPS, sep='  

Python shell

| [[1], [3], [0, 1], [1, 4], [1, 5], [1]]
| [0.039365 0.353211 0.02751 0.322593 0.1623 0.095021]
Simulate random walk (random surfer model)

```python
import matplotlib.pyplot as plt
plt.bar(range(6), count)
plt.title("Random Walk")
plt.xlabel("node")
plt.ylabel("number of visits")
plt.show()
```
Transition matrix $A$

$$A = \frac{G}{\text{degree}}$$

# Normalize row sums to one. Note that 'degree'
# is an $n \times 1$ matrix, whereas $G$ is an $n \times n$ matrix.
# The elementwise division is repeated for each column of $G$

```python
print(A)
```

```
[[0.  1.  0.  0.  0.  0. ]
 [0.  0.  0.  1.  0.  0. ]
 [0.5 0.5 0.  0.  0.  0. ]
 [0.  0.5 0.  0.  0.5 0. ]
 [0.  0.5 0.  0.  0.  0.5]
 [0.  1.  0.  0.  0.  0. ]]
```
Repeated matrix multiplication

We now want to compute the probability $p^{(i)}_j$ to be in vertex $j$ after $i$ steps. Let $p^{(i)} = (p^{(i)}_0, \ldots, p^{(i)}_{n-1})$. Initially we have $p^{(0)} = (1, 0, \ldots, 0)$.

We compute a matrix $M$, such that $p^{(i)} = M^i \cdot p^{(0)}$ (assuming $p^{(0)}$ is a column vector).

If we let $1_n$ denote the $n \times n$ matrix with 1 in each entry, then $M$ can be computed as:

$$ p_j^{(i+1)} = \frac{1}{6} \cdot \frac{1}{n} + \frac{5}{6} \sum_k p_k^{(i)} \cdot A_{k,j} $$

$$ p^{(i+1)} = \left( \frac{1}{6} \cdot \frac{1}{n} 1_n + \frac{5}{6} A^T \right) \cdot p^{(i)} $$

Python shell:

```python
pagerank.ipynb
ITERATIONS = 20
p_0 = np.zeros((n, 1))
p_0[0, 0] = 1.0
M = 1 / (6 * n) + 5 / 6 * A.T
p = p_0
prob = p  # 'prob' will contain each
# computed 'p' as a new column
for _ in range(ITERATIONS):
    p = M @ p
    prob = np.append(prob, p, axis=1)
print(p)
```

<table>
<thead>
<tr>
<th></th>
<th>[0.03935185]</th>
<th>[0.35326184]</th>
<th>[0.02777778]</th>
<th>[0.32230071]</th>
<th>[0.16198059]</th>
<th>[0.09532722]</th>
</tr>
</thead>
</table>

```
Rate of convergence

```
import matplotlib.pyplot as plt

ITERATIONS = 20
n = len(prob)  # Assume prob is a list of probabilities

x = range(ITERATIONS + 1)
for node in range(n):
    plt.plot(x, prob[node], label="node %s" % node)
plt.xticks(x)
plt.title("Random Surfer Probabilities")
plt.xlabel("Iterations")
plt.ylabel("Probability")
plt.legend()
plt.show()
```
Repeated squaring

\[
M \cdot (\cdots (M \cdot (M \cdot p(0)) \cdots)) = M^k \cdot p(0) = M^{2^{\log k}} \cdot p(0) = (\cdots ((M^2)^2)^2 \cdots)^2 \cdot p(0)
\]

\(k\) multiplications, \(k\) power of 2

```
from math import log
MP = M
for _ in range(1 + int(log(ITERATIONS, 2))):
    MP = MP @ MP
p = MP @ p_0
print(p)
```

```
[[0.03935185]
 [0.35332637]
 [0.02777778]
 [0.32221711]
 [0.16203446]
 [0.09529243]]
```
PageRank: Computing eigenvector for $\lambda = 1$

- We want to find a vector $p$, with $|p| = 1$, where $Mp = p$, i.e. an eigenvector $p$ for the eigenvalue $\lambda = 1$

```python
import numpy as np

M = np.array([[...]])  # Matrix M

eigenvalues, eigenvectors = np.linalg.eig(M)

idx = eigenvalues.argmax()  # find the largest eigenvalue (= 1)
p = np.real(eigenvectors[:, idx])  # .real returns the real part of complex numbers
p /= p.sum()  # normalize p to have sum 1

print(p)
```

```
[0.03935185 0.3533267  0.02777778 0.32221669 0.16203473 0.09529225]
```
PageRank: Note on practicality

- In practice an explicit matrix for billions of nodes is infeasible, since the number of entries would be order of $10^{18}$

- Instead use sparse matrices (in Python module scipy.sparse) and stay with repeated multiplication
Linear programming
scipy.optimize.linprog

- scipy.optimize.linprog can solve *linear programs* of the following form, where one wants to find an $n \times 1$ vector $x$ satisfying:

**Minimize:** $c^T \cdot x$

**Subject to:**

- $A_{ub} \cdot x \leq b_{ub}$
- $A_{eq} \cdot x = b_{eq}$

**Dimension**
- $c : n \times 1$
- $A_{ub} : m \times n$, $b_{ub} : m \times 1$
- $A_{eq} : k \times n$, $b_{eq} : k \times 1$

NB: For industrial strength linear solvers, use solvers like [Cplex](https://www.cplex.com) or [Gurobi](https://www.gurobi.com).
Linear programming example

**Maximize**

\[ 3 \cdot x_1 + 2 \cdot x_2 \]

Subject to

\[ 2 \cdot x_1 + 1 \cdot x_2 \leq 10 \]
\[ 5 \cdot x_1 + 6 \cdot x_2 \geq 4 \]
\[ -3 \cdot x_1 + 7 \cdot x_2 = 8 \]

**Minimize**

\[ -(3 \cdot x_1 + 2 \cdot x_2) \]

Subject to

\[ 2 \cdot x_1 + 1 \cdot x_2 \leq 10 \]
\[ -5 \cdot x_1 + -6 \cdot x_2 \leq -4 \]
\[ -3 \cdot x_1 + 7 \cdot x_2 = 8 \]

```
import numpy as np
from scipy.optimize import linprog

c = np.array([3, 2])
A_ub = np.array([[2, 1], [-5, -6]])  # multiplied by -1
b_ub = np.array([10, -4])
A_eq = np.array([[-3, 7]])
b_eq = np.array([8])
res = linprog(-c,  # maximize = minimize the negated
              A_ub=A_ub,
              b_ub=b_ub,
              A_eq=A_eq,
              b_eq=b_eq)
print(res)  # res.x is the optimal vector
```

Python shell

```
| fun:  -16.35294117647059
| message: 'Optimization terminated successfully.'
| nit:  3
| slack: array([0. , 30.47058824])
| status: 0
| success: True
| x:  array([3.64705882, 2.70588235])
```
Solving maximum flow using linear programming

We will use the `scipy.optimize.linprog` function to solve the maximum flow problem on the above directed graph. We want to send as much flow from node A to node F. Edges are numbered 0..8 and each edge has a maximum capacity.

Note: solution not unique
Solving maximum flow using linear programming

- \( \mathbf{x} \) is a vector describing the flow along each edge
- \( \mathbf{c} \) is a vector that to add the flow along the edges (7 and 8) to the sink (F), i.e. a function computing the flow value
- \( \mathbf{A_{ub}} \) and \( \mathbf{b_{ub}} \) is a set of capacity constraints, for each edge flow \( \leq \) capacity
- \( \mathbf{A_{eq}} \) and \( \mathbf{b_{eq}} \) is a set of flow conservation constraints, for each non-source and non-sink node (B, C, D, E), requiring that the flow into equals the flow out of a node

Maximize
\[
\mathbf{c}^T \cdot \mathbf{x}
\]
Subject to
\[
\begin{align*}
\mathbf{A_{ub}} \cdot \mathbf{x} & \leq \mathbf{b_{ub}} \\
\mathbf{A_{eq}} \cdot \mathbf{x} & = \mathbf{b_{eq}} = \mathbf{0}
\end{align*}
\]
\( \mathbf{I} \cdot \mathbf{x} \leq \text{capacity} \)

\[
\begin{align*}
x_0 & \leq 4 \\
x_1 & \leq 3 \\
x_2 & \leq 1 \\
x_3 & \leq 1 \\
x_4 & \leq 3 \\
x_5 & \leq 1 \\
x_6 & \leq 3 \\
x_7 & \leq 1 \\
x_8 & \leq 5
\end{align*}
\]
import numpy as np
from scipy.optimize import linprog

# 0 1 2 3 4 5 6 7 8
conservation = np.array([[0, -1, 0, 0, 1, 1, 0, 0, 0],  # B
                         [-1, 0, 1, 1, 0, 0, 0, 0, 0],  # C
                         [0, 0, 0, -1, 0, -1, -1, 0, 1],  # D
                         [0, 0, -1, 0, -1, 0, 1, 1, 0]])  # E

# 0 1 2 3 4 5 6 7 8
sinks = np.array([0, 0, 0, 0, 0, 0, 0, 1, 1])

# 0 1 2 3 4 5 6 7 8
capacity = np.array([4, 3, 1, 1, 3, 1, 3, 1, 5])

res = linprog(-sinks,
              A_eq=conservation,
              b_eq=np.zeros(conservation.shape[0]),
              A_ub=np.eye(capacity.size),
              b_ub=capacity)

print(res)

the solution found varies with the scipy version
Generators, iterators

- `__iter__`, `__next__`
- `yield`
- `generator expression`
- `measuring memory usage`
Iterable & Iterator

### Python shell
```
> L = ['a', 'b', 'c']
> type(L)  
  | <class 'list'>
> it = L.__iter__()
> type(it)  
  | <class 'list_iterator'>
> it.__next__()  
  | 'a'
> it.__next__()  
  | 'b'
> it.__next__()  
  | 'c'
> it.__next__()  
  | StopIteration # Exception
```

### Python shell
```
> L = ['a', 'b', 'c']
> it = iter(L)  # calls L.__iter__()
> next(it)       # calls it.__next__()
  | 'a'
> next(it)       # calls it.__next__()
  | 'b'
> next(it)       # calls it.__next__()
  | 'c'
> next(it)       # calls it.__next__()
  | StopIteration
```

- Lists are **iterable** (must support `__iter__`)
- `iter` returns an **iterator** (must support `__next__`)

Some iterables in Python: string, list, set, tuple, dict, range, enumerate, zip, map, reversed
Iterator

- `next(iterator_object)` returns the next element from the iterator, by calling the `iterator_object.__next__()` If no more elements to be report raise exception `StopIteration`

- `next(iterator_object, default)` returns `default` when no more elements are available (no exception is raised)

- for-loops and list comprehensions require iterable objects
  ```python
  for x in range(5): and  [2**x for x in range(5)]
  ```

- The iterator concept is also central to Java and C++
# for loop

<table>
<thead>
<tr>
<th>Python shell</th>
<th>Python shell</th>
</tr>
</thead>
</table>
| > for x in ['a', 'b', 'c']:
  print(x)  |
|   | a   |
|   | b   |
|   | c   |
| iter (can call iter on it to generate an iterator) | > L = ['a', 'b', 'c']
| | > it = iter(L) |
| | > while True:
| |   try:
| |     x = next(it)
| |   except StopIteration:
| |     break
| | print(x)  |
| | a   |
| | b   |
| | c   |
8.3. The **for** statement

The **for** statement is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object:

```python
for_stmt ::=  "for"  target_list "in"  expression_list  "::"  suite
             ["else"  "::"  suite]
```

The expression list is evaluated once; it should yield an iterable object. An iterator is created for the result of the `expression_list`. The suite is then executed once for each item provided by the iterator, in the order returned by the iterator. Each item in turn is assigned to the target list using the standard rules for assignments (see Assignment statements), and then the suite is executed. When the items are exhausted (which is immediately when the sequence is empty or an iterator raises a `StopIteration` exception), the suite in the `else` clause, if present, is executed, and the loop terminates.
for loop over changing iterable

Changing (extending) the list while scanning

The iterator over a list is just an index into the list

<table>
<thead>
<tr>
<th>Python shell</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>L = [1, 2]</code></td>
</tr>
<tr>
<td><code>for x in L:</code></td>
</tr>
<tr>
<td><code>print(x, L)</code></td>
</tr>
<tr>
<td><code>L.append(x + 2)</code></td>
</tr>
<tr>
<td>1 [1, 2]</td>
</tr>
<tr>
<td>2 [1, 2, 3]</td>
</tr>
<tr>
<td>3 [1, 2, 3, 4]</td>
</tr>
<tr>
<td>4 [1, 2, 3, 4, 5]</td>
</tr>
<tr>
<td>5 [1, 2, 3, 4, 5, 6]</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Python shell</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>L = [1, 2]</code></td>
</tr>
<tr>
<td><code>for x in L:</code></td>
</tr>
<tr>
<td><code>print(x, L)</code></td>
</tr>
<tr>
<td><code>L[:0] = [L[0] - 2, L[0] - 1]</code></td>
</tr>
<tr>
<td>1 [1,2]</td>
</tr>
<tr>
<td>0 [-1,0,1,2]</td>
</tr>
<tr>
<td>-1 [-3,-2,-1,0,1,2]</td>
</tr>
<tr>
<td>-2 [-5,-4,-3,-2,-1,0,1,2]</td>
</tr>
<tr>
<td>-3 [-7,-6,-5,-4,-3,-2,-1,0,1,2]</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
# range

<table>
<thead>
<tr>
<th>Python shell</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r = range(1, 6)</code> # 1,2,3,4,5</td>
</tr>
<tr>
<td><code>type(r)</code></td>
</tr>
<tr>
<td><code>&lt;class 'range'&gt;</code></td>
</tr>
<tr>
<td><code>it = iter(r)</code></td>
</tr>
<tr>
<td><code>type(it)</code></td>
</tr>
<tr>
<td><code>&lt;class 'range_iterator'&gt;</code></td>
</tr>
<tr>
<td><code>next(it)</code></td>
</tr>
<tr>
<td><code>1</code></td>
</tr>
<tr>
<td><code>next(it)</code></td>
</tr>
<tr>
<td><code>2</code></td>
</tr>
<tr>
<td><code>for x in it:</code></td>
</tr>
<tr>
<td><code>print(x)</code></td>
</tr>
<tr>
<td><code>3</code></td>
</tr>
<tr>
<td><code>4</code></td>
</tr>
<tr>
<td><code>5</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Python shell</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>it</code></td>
</tr>
<tr>
<td><code>&lt;range_iterator object at 0x03E7FFC8&gt;</code></td>
</tr>
<tr>
<td><code>iter(it)</code></td>
</tr>
<tr>
<td><code>&lt;range_iterator object at 0x03E7FFC8&gt;</code></td>
</tr>
<tr>
<td><code>it is iter(it)</code></td>
</tr>
<tr>
<td><code>True</code></td>
</tr>
</tbody>
</table>

Calling `iter` on a `range_iterator` just returns the iterator itself, i.e. can use the iterator wherever an iterable is expected.
Creating an iterable class

```python
# names.py

class Names:
    def __init__(self, *arg):
        self.people = arg

    def __iter__(self):
        return Names_iterator(self)

class Names_iterator:
    def __init__(self, names):
        self.idx = 0
        self.names = names

    def __next__(self):
        if self.idx >= len(self.names.people):
            raise StopIteration
        self.idx += 1
        return self.names.people[self.idx - 1]

duckburg = Names('Donald', 'Goofy', 'Mickey', 'Minnie')
for name in duckburg:
    print(name)
```

Python shell:
```
Donald
Goofy
Mickey
Minnie
```
An infinite iterable

```python
class infinite_range:
    def __init__(self, start=0, step=1):
        self.start = start
        self.step = step
    def __iter__(self):
        return infinite_range_iterator(self)

class infinite_range_iterator:
    def __init__(self, inf_range):
        self.range = inf_range
        self.current = self.range.start
    def __next__(self):
        value = self.current
        self.current += self.range.step
        return value
    def __iter__(self):
        # make iterator iterable
        return self

r = infinite_range(42, -3)
for idx, value in zip(range(5), it):
    print(idx, value)
```

```python
defining Python shell
>>> r = infinite_range(42, -3)
>>> it = iter(r)
>>> for idx, value in zip(range(5), it):
...     print(idx, value)
0 42
1 39
2 36
3 33
4 30
```

```python
>>> for idx, value in zip(range(5), it):
...     print(idx, value)
0 27
1 24
2 21
3 18
4 15
```

```python
>>> print(sum(r))  # don't do this
(sum and zip take iterables
(zip stops when shortest iterable is exhausted)(runs forever)
```
Creating an iterable class (iterable = iterator)

```python
class my_range:
    def __init__(self, start, end, step):
        self.start = start
        self.end = end
        self.step = step
        self.x = start

    def __iter__(self):
        return self  # self also iterator

    def __next__(self):
        if self.x >= self.end:
            raise StopIteration
        answer = self.x
        self.x += self.step
        return answer

r = my_range(1.5, 2.0, 0.1)
```

Python shell

```
> list(r)
[1.5, 1.6, 1.7000000000000002, 1.8000000000000003, 1.9000000000000004]
> list(r)
[]
```

- Note that objects act both as an iterable and an iterator
- This e.g. also applies to `zip` objects
- Can only iterate over a `my_range` once
# itertools

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count(start, step)</code></td>
<td>Infinite sequence: <code>start, start + step, ...</code></td>
</tr>
<tr>
<td><code>cycle(seq)</code></td>
<td>Infinite repeats of the elements from <code>seq</code></td>
</tr>
<tr>
<td><code>repeat(value[, times])</code></td>
<td>Infinite repeats of <code>value</code> or <code>times</code> repeats</td>
</tr>
<tr>
<td><code>chain(seq0,...,seqk)</code></td>
<td>Concatenate sequences</td>
</tr>
<tr>
<td><code>starmap(func, seq)</code></td>
<td></td>
</tr>
<tr>
<td><code>permutations(seq)</code></td>
<td>Generate all possible permutations of <code>seq</code></td>
</tr>
<tr>
<td><code>islice(seq, start, stop, step)</code></td>
<td>Create a slice of <code>seq</code></td>
</tr>
</tbody>
</table>

[https://docs.python.org/3/library/itertools.html](https://docs.python.org/3/library/itertools.html)
In Java iteration does not stop using exceptions, but instead the iterator can be tested if it is at the end of the iterable.
Example: C++ iterators

```cpp
#include <iostream>
#include <vector>

int main() {
    // Vector is part of STL (Standard Template Library)
    std::vector<int> A = {20, 23, 26};
    // "C" indexing - since C++98
    for (int i = 0; i < A.size(); i++)
        std::cout << A[i] << std::endl;
    // Iterator - since C++98
    for (std::vector<int>::iterator it = A.begin(); it != A.end(); ++it)
        std::cout << *it << std::endl;
    // "auto" iterator - since C++11
    for (auto it = A.begin(); it != A.end(); ++it)
        std::cout << *it << std::endl;
    // Range-based for-loop - since C++11
    for (auto e : A)
        std::cout << e << std::endl;
}
```

In C++ iterators can be tested if they reach the end of the iterable.

Move iterator to next element.
Generators
Generator expressions

A generator expression

\[(\ldots \text{for } x \text{ in } \ldots)\]

looks like a list comprehension, except square brackets are replaced by parenthesis

Is an iterable and iterator, that uses less memory than a list comprehension

Computation is done \textit{lazily}, i.e. first when needed

Python shell

```python
>>> [x**2 for x in range(5)]  # list comprehension
| [0, 1, 4, 9, 16]  # list
>>> (x**2 for x in range(3))  # generator expression
| <generator object <genexpr> at 0x03D9F8A0>
>>> o = (x**2 for x in range(3))
>>> next(o)
| 0
>>> next(o)
| 1
>>> next(o)
| 4
>>> next(o)
| StopIteration
```

https://docs.python.org/3/reference/expressions.html#generator-expressions
Nested generator expressions

- Each fraction is first computed when requested by `next(ratios)` (implicitly called repeatedly in `list(ratios)`)
- The next value of `squares` is first computed when needed by `ratios`
Generator expressions as function arguments

- Python allows to omit a pair of parenthesis when a generator expression is the only argument to a function

\[
 f(\ldots \text{ for } x \text{ in } \ldots) \equiv f((\ldots \text{ for } x \text{ in } \ldots))
\]

Python shell

```python
squares = (x*2 for x in range(1, 6))
sum(squares)  # sum takes an iterable
30
sum((x*2 for x in range(1, 6)))
30
sum(x*2 for x in range(1, 6))  # one pair of parenthesis omitted
30
```

PEP 289 – Generator Expressions
Generator functions

A generator function contains one or more `yield` statements

Python automatically makes a call to a generator function into an iterable and iterator (provides `__iter__` and `__next__`)

Calling a generator function returns a generator object

Whenever `next` is called on a generator object, the executing of the function continues until the next `yield exp` and the value of `exp` is returned as a result of `next`

Reaching the end of the function or a return statement, will raise `StopIteration`

Once consumed, can't be reused

```
two.py

def two():
    yield 1
    yield 2
```

Python shell

```
> two()
<generator object two at 0x03629510>
> t = two()
> next(t)
1
> next(t)
2
> next(t)
StopIteration
```

https://docs.python.org/3/reference/expressions.html#yield-expressions
Generator functions (II)

```python
my_generator.py
def my_generator(n):
    yield 'Start'
    for i in range(n):
        yield chr(ord('A') + i)
    yield 'Done'

Python shell
> g = my_generator(3)
> print(g)
  <generator object my_generator at 0x03E2F6F0>
> print(list(g))
  ['Start', 'A', 'B', 'C', 'Done']
> print(list(g))  # generator object g exhausted
  []
> print(*my_generator(5))  # * takes an iterable (PEP 448)
  Start A B C D E Done
Generator functions (III)

```python
my_range_generator.py

def my_range(start, end, step):
    x = start
    while x < end:
        yield x
        x += step

Python shell

> list(my_range(1.5, 2.0, 0.1))
[1.5, 1.6, 1.7000000000000002, 1.8000000000000003, 1.9000000000000004]
```
Pipelining generators

```python
# Python shell
> def squares(seq):  # seq should be an iterable object
    for x in seq:    # use iterator to run through seq
        yield x**2     # generator
> list(squares(range(5)))
| [0, 1, 4, 9, 16]
> list(squares(squares(range(5))))  # pipelining generators
| [0, 1, 16, 81, 256]
> sum(squares(squares(range(100000000))))  # pipelining generators
| 19999999500000000333333333333333330000000
> sum((x**2)**2 for x in range(100000000))  # generator expression
| 19999999500000000333333333333333330000000
> sum([(x**2)**2 for x in range(100000000)])  # list comprehension
| MemoryError
```
yield vs yield from

Python shell
```
> def g():
    yield 1
    yield [2,3,4]
    yield 5
> list(g())
[1, [2, 3, 4], 5]
```

Python shell
```
> def g():
    yield 1
    yield from [2,3,4]
    yield 5
> list(g())
[1, 2, 3, 4, 5]
```

- `yield from` available since Python 3.3
- `yield from exp` ≈ `for x in exp: yield x`
Recursive `yield from`

```python
def traverse(T):
    # recursive generator
    if isinstance(T, tuple):
        for child in T:
            yield from traverse(child)
    else:
        yield T

T = (((1,2),3,(4,5)),(6,(7,9)))
traverse(T)
```

```bash
<generator object traverse at 0x03279F30>
```

```
[1, 2, 3, 4, 5, 6, 7, 9]
```
Making objects iterable using `yield`

```python
vector2D.py

class vector2D:
    def __init__(self, x_value, y_value):
        self.x = x_value
        self.y = y_value
    def __iter__(self):
        # generator
        yield self.x
        yield self.y
    def __iter__(self):
        # alternative generator
        yield from (self.x, self.y)

v = vector2D(5, 7)
print(list(v))
print(tuple(v))
print(set(v))
```

Python shell
```
[5, 7]
(5, 7)
{5, 7}
```
Generators vs iterators

- Iterators can often be reused (can copy the current state)
- Generators cannot be reused (only if a new generator is created, starting over again)

- David Beazley’s tutorial on “Generators: The Final Frontier”, PyCon 2014 (3:50:54)
  Throughout advanced discussion of generators, e.g. how to use `.send` method to implement coroutines
  https://www.youtube.com/watch?v=D1twn9kLmYg
Measuring memory usage
Measuring memory usage (memory profiling)

- Macro level:
  - Task Manager (Windows)
  - Activity Monitor (Mac)
  - top (Linux)

- Variable level:
  - `getsizeof` from `sys` module

- Detailed overview:
  - Module `memory_profiler`
    - Allows detailed space usage of the code line-by-line (using @profile function decorator) or a plot of total space usage over time
    - `pip install memory-profiler`

Python shell
```
> import sys
> sys.getsizeof(42)
| 14  # size of the integer 42 is 14 bytes
> sys.getsizeof(42**42)
| 44  # the size increases with value
> sys.getsizeof('42')
| 27  # size of a string
> import numpy as np
> sys.getsizeof(np.array(range(100), dtype='int32'))
| 448  # also works on Numpy arrays
> squares = [x**2 for x in range(1000000)]
> sys.getsizeof(squares)
| 4348736
> g = (x**2 for x in range(1000000))
> sys.getsizeof(g)
| 64
```
Module
memory-profiler
pypi.org/project/memory-profiler/

memory_usage.py

from memory_profiler import profile

@profile  # prints new statistics for each call
def use_memory():
    s = 0
    x = list(range(20_000_000))
    s += sum(x)
    y = list(range(10_000_000))
    s += sum(x)

use_memory()

Python Shell

Filename: C:/.../memory_usage.py

<table>
<thead>
<tr>
<th>Line #</th>
<th>Mem usage</th>
<th>Increment</th>
<th>Line Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>32.0 MiB</td>
<td>32.0 MiB</td>
<td>@profile</td>
</tr>
<tr>
<td>4</td>
<td>32.0 MiB</td>
<td>0.0 MiB</td>
<td>def use_memory():</td>
</tr>
<tr>
<td>5</td>
<td>32.0 MiB</td>
<td>0.0 MiB</td>
<td>s = 0</td>
</tr>
<tr>
<td>6</td>
<td>415.9 MiB</td>
<td>383.9 MiB</td>
<td>x = list(range(20_000_000))</td>
</tr>
<tr>
<td>7</td>
<td>415.9 MiB</td>
<td>0.0 MiB</td>
<td>s += sum(x)</td>
</tr>
<tr>
<td>8</td>
<td>607.8 MiB</td>
<td>191.9 MiB</td>
<td>y = list(range(10_000_000))</td>
</tr>
<tr>
<td>9</td>
<td>607.8 MiB</td>
<td>0.0 MiB</td>
<td>s += sum(x)</td>
</tr>
</tbody>
</table>

memory_sin_usage.py

from math import sin, pi

for a in range(1000):
    x = list(range(int(1000000 * sin(pi * a / 250)))))

Windows Shell

> pip install memory-profiler
> mprof run memory_sin_usage.py
| mprof: Sampling memory every 0.1s  | running as a Python program... |
> mprof plot
Modules and packages

- `import` – `from` – `as`
- `__name__`, "__main__"

docs.python.org/3/tutorial/modules.html
Python modules and packages

- A Python module is a `module_name`.py file containing Python code
- A Python package is a collection of modules

Why do you need modules?
- A way to structure code into smaller logical units
- Encapsulation of functionality
- Reuse of code in different programs

- Your can write your own modules and packages or use any of the +100,000 existing packages from pypi.org
- The Python Standard Library consists of the modules listed on docs.python.org/3/library
A module is only run once when imported several times
Some modules mentioned in the course

<table>
<thead>
<tr>
<th>Module (example functions)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>math (pi sqrt ceil log sin)</td>
<td>basic math</td>
</tr>
<tr>
<td>random (random randint)</td>
<td>random number generator</td>
</tr>
<tr>
<td>numpy (array shape)</td>
<td>multi-dimensional data</td>
</tr>
<tr>
<td>pandas</td>
<td>data tables</td>
</tr>
<tr>
<td>SQLite</td>
<td>SQL database</td>
</tr>
<tr>
<td>scipy</td>
<td>mathematical optimization</td>
</tr>
<tr>
<td>scipy.optimize (minimize linprog)</td>
<td></td>
</tr>
<tr>
<td>scipy.spatial (ConvexHull)</td>
<td></td>
</tr>
<tr>
<td>matplotlib</td>
<td>plotting data</td>
</tr>
<tr>
<td>matplotlib.pyplot</td>
<td>print plots to PDF</td>
</tr>
<tr>
<td>matplotlib.backends.backend_pdf</td>
<td>3D plot tools</td>
</tr>
<tr>
<td>doctest (testmod)</td>
<td>testing using doc strings</td>
</tr>
<tr>
<td>unittest</td>
<td>unit testing</td>
</tr>
<tr>
<td>time (time)</td>
<td>current time, conversion of time values</td>
</tr>
<tr>
<td>datetime (date.today)</td>
<td></td>
</tr>
<tr>
<td>timeit (timeit)</td>
<td>time execution of simple code</td>
</tr>
<tr>
<td>heapq</td>
<td>use a list as a heap</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Module (example functions)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>functools</td>
<td>higher order functions and decorators</td>
</tr>
<tr>
<td>itertools</td>
<td>Iterator tools</td>
</tr>
<tr>
<td>collections (Counter deque)</td>
<td>data structures for collections</td>
</tr>
<tr>
<td>os (path)</td>
<td>operating system interface</td>
</tr>
<tr>
<td>sys (argv path)</td>
<td>system specific functions</td>
</tr>
<tr>
<td>Tkinter</td>
<td>graphic user interface</td>
</tr>
<tr>
<td>PyQt</td>
<td></td>
</tr>
<tr>
<td>xml</td>
<td>xml files (eXtensible Markup Language)</td>
</tr>
<tr>
<td>json</td>
<td>JSON (JavaScript Object Notation) files</td>
</tr>
<tr>
<td>csv</td>
<td>comma separated files</td>
</tr>
<tr>
<td>openpyxl</td>
<td>EXCEL files</td>
</tr>
<tr>
<td>re</td>
<td>regular expression, string searching</td>
</tr>
<tr>
<td>string (split join lower ascii_letters digits)</td>
<td>string functions</td>
</tr>
</tbody>
</table>
Ways of importing modules

```
# import.py

# Import a module name in the current namespace
# All definitions in the module are available as <module>.<name>
import math
print(math.sqrt(2))

# Import only one or more specific definitions into current namespace
from math import sqrt, log, ceil
print(ceil(log(sqrt(100), 2)))

# Import specific modules/definitions from a module into current namespace under new names
from math import sqrt as kvadratrod, log as logaritme
import matplotlib.pyplot as plt
print(logaritme(kvadratrod(100)))

# Import all definitions from a module in current namespace
# Deprecated, since unclear what happens to the namespace
from math import *
print(pi)  # where did 'pi' come from?
```

Python shell

```
from math import sqrt
appears to be faster than math.sqrt

```
sqrt_performance.py

from time import time
import math
start = time()
x = sum(math.sqrt(x) for x in range(10000000))
end = time()
print("math.sqrt", end - start)

from math import sqrt
start = time()
x = sum(sqrt(x) for x in range(10000000))
end = time()
print("from math import sqrt", end - start)

def test(sqrt=math.sqrt):
    # abuse of keyword argument
    start = time()
x = sum(sqrt(x) for x in range(10000000))
end = time()
print("bind sqrt to keyword argument", end - start)

test()
```

Python shell

```
| math.sqrt      4.05187726020813 |
| from math import sqrt  3.5011463165283203 |
| bind sqrt to keyword argument  3.261594772338867 |
```
Listing definitions in a module: \texttt{dir(module)}

\begin{Verbatim}
Python shell

\texttt{> import math}
\texttt{> import matplotlib.pyplot as plt}
\texttt{> dir(math)}
\texttt{> help(math)}
\texttt{| Help on built-in module math:}
\texttt{| NAME}
\texttt{| math}
\texttt{| DESCRIPTION}
\texttt{| ...}
\end{Verbatim}
The variable `__name__` contains the name of the module, or `__main__` if the file is run as the main file by the interpreter.

Can e.g. be used to test a module if the module is run independently.
module `importlib`

- Implements the `import` statement (Python internal implementation details)
- `importlib.reload(module)`
  - Reloads a previously imported `module`. Relevant if you have edited the code for the module and want to load the new version in the Python interpreter, without restarting the full program from scratch.
Packages

- A package is a collection of modules (and subpackages) in a folder = package name
- Only folders having an `__init__.py` file are considered packages
- The `__init__.py` can be empty, or contain code that will be loaded when the package is imported, e.g. importing specific modules

```python
mymodule/__init__.py

mymodule/a.py
print("Loading mymodule.a")
def f():
    print("mymodule.a.f")

using_mymodule.py
import mymodule.a
mymodule.a.f()

Python shell
| Loading mymodule.a
| mymodule.a.f
```
### A package with a subpackage

<table>
<thead>
<tr>
<th>mypackage/<strong>init</strong>.py</th>
<th>using_mysubpackage.py</th>
</tr>
</thead>
<tbody>
<tr>
<td>print('loading mypackage')</td>
<td>import mypackage.a</td>
</tr>
<tr>
<td>mypackage/a.py</td>
<td>mypackage.a.f()</td>
</tr>
<tr>
<td>print('Loading mypackage.a')</td>
<td>import mypackage.mysubpackage</td>
</tr>
<tr>
<td>def f():</td>
<td>mypackage.mysubpackage.b.g()</td>
</tr>
<tr>
<td>print('mypackage.a.f')</td>
<td>from mypackage.mysubpackage.b import g</td>
</tr>
<tr>
<td>mypackage/mysubpackage/<strong>init</strong>.py</td>
<td>g()</td>
</tr>
<tr>
<td>print('loading mypackage.mysubpackage')</td>
<td>Python shell</td>
</tr>
<tr>
<td>import mypackage.mysubpackage.b</td>
<td></td>
</tr>
<tr>
<td>mypackage/mysubpackage/b.py</td>
<td>loading mypackage</td>
</tr>
<tr>
<td>print('Loading mypackage.mysubpackage.b')</td>
<td>Loading mypackage.a</td>
</tr>
<tr>
<td>def g():</td>
<td>mypackage.a.f</td>
</tr>
<tr>
<td>print('mypackage.mysubpackage.b.g')</td>
<td>loading mypackage.mysubpackage</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>loading mypackage.mysubpackage.b</td>
</tr>
<tr>
<td></td>
<td>mypackage.mysubpackage.b.g</td>
</tr>
<tr>
<td></td>
<td>mypackage.mysubpackage.b.g</td>
</tr>
</tbody>
</table>
When Python loads a module the first time it is compiled to some intermediate code, and stored as a .pyc file in the `__pycache__` folder.

If a .pyc file exists for a module, and the .pyc file is newer than the .py file, then `import` loads .pyc – saving time to load the module (but does not make the program itself faster).

It is safe to delete the `__pycache__` folder – but it will be created again next time a module is loaded.
Path to modules

Python searches the following folders for a module in the following order:

1) The directory containing the input script / current directory
2) Environment variable PYTHONPATH
3) Installation defaults

The function path in the module sys returns a list of the paths
Setting PYTHONPATH from windows shell

- set PYTHONPATH=
  paths separated by semicolon
  (only valid until shell is closed)
Setting PYTHONPATH from control panel

- Control panel > System > Advanced system settings > Environment Variables > User variables > Edit or New PYTHONPATH
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
module `heapq` (Priority Queue)

- Implements a binary heap (Williams 1964).
- Stores a set of elements in a standard list, where arbitrary elements can be inserted efficiently and the smallest element can be extracted efficiently

```
import heapq
from random import random

H = []  # a heap is just a list
for _ in range(10):
    heapq.heappush(H, random())

while True:
    x = heapq.heappop(H)
    print(x)
    heapq.heappush(H, x + random())
```

Python shell
```
0.20569933892764458
0.27057819339616174
0.3115615362876237
0.4841062272152259
0.5054280956005357
0.509387117524076
0.598647195480462
0.7035150735555027
0.7073929685826221
0.7091224012815325
0.714213496127318
0.727868481291271
0.8051275413759873
0.8279523767282903
0.8626022363202895
0.937663123626869
```

docs.python.org/3/library/heapq.html
Valid heap

- A valid heap satisfies for all i: 
  \[L[i] \leq L[2 \cdot i + 1] \text{ and } L[i] \leq L[2 \cdot i + 2]\]

- `heapify(L)` rearranges the elements in a list to make the list a valid heap

```
Python shell

from random import randint
L = [randint(1, 20) for _ in range(10)]
L  # just random numbers
[18, 1, 15, 17, 4, 14, 11, 3, 4, 9]
import heapq
heapq.heapify(L)  # make L a valid heap
L
[1, 3, 11, 4, 4, 14, 15, 17, 18, 9]
print(heapq.heappop(L))
1
L
[3, 4, 11, 4, 9, 14, 15, 17, 18]
heapq.heappush(L, 7)
L
[3, 4, 11, 4, 7, 14, 15, 17, 18, 9]
```
Why `heapq`?

- `min` and `remove` on a list take *linear time* (runs through the whole list)
- `heapq` supports `heappush` and `heappop` in *logarithmic time*
- For lists of length 30,000,000 the performance gain is a factor 200,000
```python
import heapq
from random import random
import matplotlib.pyplot as plt
from time import time
import gc  # garbage collection

size = []
time_heap = []
time_list = []

for i in range(26):
    n = 2 ** i
    size.append(n)
    L = [random() for _ in range(n)]
    R = max(1, 2 ** 23 // n)
    gc.collect()
    start = time()
    for _ in range(R):
        L.append(random())
        x = min(L)
        L.remove(x)
    end = time()
    time_list.append((end - start) / R)
    L = None  # avoid MemoryError
    L = [random() for _ in range(n)]
    heapq.heapify(L)  # make L a legal heap
    for _ in range(100000):
        heapq.heappush(L, random())
        x = heapq.heappop(L)
    end = time()
    time_heap.append((end - start) / 100000)

plt.title("Average time for insert + delete min")
plt.xlabel("list size")
plt.ylabel("time (seconds)"
plt.plot(size, time_list, 'b.-', label='list (append, min, remove)')
plt.plot(size, time_heap, 'r.-', label='heapq (heappush, heappop)')
plt.xscale('log')
plt.yscale('log')
plt.legend()
plt.show()
```

A: Avoid out of memory error for largest experiment, by allowing old `L` to be garbage collected

B: Reduce noise in experiments by forcing Python garbage collection before measurement
Working with text

- file formats
- CSV, JSON, XML, Excel
- regular expressions
- module re, finditer
Some file formats

<table>
<thead>
<tr>
<th>File extension</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>.html</td>
<td>HyperText Markup Language</td>
</tr>
<tr>
<td>.mp3</td>
<td>Audio File</td>
</tr>
<tr>
<td>.png, .jpeg, .jpg</td>
<td>Image files</td>
</tr>
<tr>
<td>.svg</td>
<td>Scalable Vector Graphics file</td>
</tr>
<tr>
<td>.json</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>.csv</td>
<td>Comma separated values</td>
</tr>
<tr>
<td>.xml</td>
<td>eXtensible Markup Language</td>
</tr>
<tr>
<td>.xlsx</td>
<td>Microsoft Excel 2010/2007 Workbook</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.exe</td>
<td>Windows executable file</td>
</tr>
<tr>
<td>.app</td>
<td>Max OS X Application</td>
</tr>
<tr>
<td>.py</td>
<td>Python program</td>
</tr>
<tr>
<td>.pyc</td>
<td>Python compiled file</td>
</tr>
<tr>
<td>.java</td>
<td>Java program</td>
</tr>
<tr>
<td>.cpp</td>
<td>C++ program</td>
</tr>
<tr>
<td>.c</td>
<td>C program</td>
</tr>
<tr>
<td>.txt</td>
<td>Raw text file</td>
</tr>
</tbody>
</table>
PIL – the Python Imaging Library

- pip install Pillow

```
rotate_image.py
from PIL import Image
img = Image.open("Python-Logo.png")
img_out = img.rotate(45, expand=True)
img_out.save("Python-rotated.png")
```

- For many file types there exist Python packages handling such files, e.g. for images Pillow supports 40+ different file formats

- For more advanced computer vision tasks you should consider OpenCV

[python-pillow.org](https://python-pillow.org)
CSV files - Comma Separated Values

- Simple 2D tables are stored as rows in a file, with values separated by comma
- Strings stored are quoted if necessary
- Values read are strings
- The delimiter (default comma) can be changed by keyword argument delimiter. Other typical delimiters are tabs '\t', and semicolon ';'"
CSV files - Tab Separated Values

```python
import csv

FILE = 'tab-separated.csv'

with open(FILE) as infile:
    for row in csv.reader(infile, delimiter='\t'):
        print(row)
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
```
Reading an Excel generated CSV file

average.py

```python
import csv

with open('grades.csv') as file:
    data = csv.reader(file, delimiter=';')  # data = iterator over the rows
    header = next(data)  # ['Name', 'Course', 'Grade']
    count = {}
    total = {}
    for row in data:
        # iterate over data rows
        course = row[header.index('Course')]
        grade = int(row[header.index('Grade')])
        count[course] = count.get(course, 0) + 1
        total[course] = total.get(course, 0) + grade

print('Average grades:')
width = max(map(len, count))  # maximum course name length
for course in count:
    print(f'{course:{width}s} : {total[course] / count[course]:.2f}')
```

Python shell

```
<table>
<thead>
<tr>
<th>Average grades:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis : 1.67</td>
</tr>
<tr>
<td>Programming : 1.50</td>
</tr>
<tr>
<td>Statistics : 2.50</td>
</tr>
</tbody>
</table>
```
CSV files
- Quoting

- The amount of quoting is controlled with keyword argument **quoting**
- `csv.QUOTE_MINIMAL` etc. can be used to select the quoting level
- Depending on choice of quoting, numeric values and strings cannot be distinguished in CSV file (`csv.reader` will read all as strings anyway)

```python
import csv
import sys
data = [[1, 1.0, '1.0'], ['abc', '', '\t', ',']]
quoting_options = [(csv.QUOTE_MINIMAL, "QUOTE_MINIMAL"),
                    (csv.QUOTE_ALL, "QUOTE_ALL"),
                    (csv.QUOTE_NONNUMERIC, "QUOTE_NONNUMERIC"),
                    (csv.QUOTE_NONE, "QUOTE_NONE")]
for quoting, name in quoting_options:
    print(name)
    csv_out = csv.writer(sys.stdout, quoting=quoting, escapechar='\')
    for row in data:
        csv_out.writerow(row)
```

**Python shell**

```
QUOTE_MINIMAL  # cannot distinguish 1.0 and "1.0"
1,1.0,1.0
"abc","","",""
QUOTE_ALL      # cannot distinguish 1.0 and "1.0"
"1","1.0","1.0"
"abc","","",""
QUOTE_NONNUMERIC
1,1.0,"1.0"
"abc","","",""
QUOTE_NONE     # cannot distinguish 1.0 and "1.0"
1,1.0,1.0
"abc","","\\"",""'
```
File encodings...

- Text files can be *encoded* using many different encodings (UTF-8, UTF-16, UTF-32, Windows-1252, ANSI, ASCII, ISO-8859-1, ...)
- Different encodings can result in different file sizes, in particular when containing non-ASCII symbols
- Programs often try to predict the encoding of text files (often with success, but not always)
- Opening files assuming wrong encoding can give strange results....

[Files and encodings example]

Opening UTF-8 encoded file but trying to decode using Windows-1252

Opening Windows-1252 encoded file but trying to decode using UTF-8

[Source: en.wikipedia.org/wiki/Character_encoding]
for filename in ['river-utf8.txt', 'river-windows1252.txt']:
    print(filename)
    f = open(filename, 'rb')  # open input in binary mode, default = text mode = 't'
    line = f.readline()  # type(line) = bytes = immutable list of integers in 0..255
    print(line)  # byte literals look like strings, prefixed 'b'
    print(list(line))  # print bytes as list of integers
    f = open(filename, 'r', encoding='utf-8')  # try to open file as UTF-8
    line = f.readline()  # fails if input line is not utf-8
    print(line)

Python shell
---

```
| river-utf8.txt
| b'\xc3\x86 \xc3\x86 U I \xc3\x86 \xc3\x85\r\n'  # \x = hexadecimal value follows
| [195, 134, 32, 195, 134, 32, 85, 32, 73, 32, 195, 134, 32, 195, 133, 13, 10]
| E E U I E Å

| river-windows1252.txt
| b'\xc6 \xc6 U I \xc6 \xc5\r\n'
| [198, 32, 198, 32, 85, 32, 73, 32, 198, 32, 197, 13, 10]
| UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc6 in position 0: invalid continuation byte

> 'E E U I E Å'.encode('utf8')  # convert string to (an immutable array of) bytes
| b'\xc3\x86 \xc3\x86 U I \xc3\x86 \xc3\x85'

> 'E E U I E Å'.encode('utf8').decode('Windows-1252')  # decode bytes to string
| 'Å† Å† U I Å† Å…'
```
Reading CSV files with specific encoding

```python
dontrast
# read_shopping.py
import csv

with open("shopping.csv", encoding="Windows-1252") as file:
    for article, amount in csv.reader(file):
        print("Buy", amount, article)
```

```
<table>
<thead>
<tr>
<th>Python shell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buy 2 æbler</td>
</tr>
<tr>
<td>Buy 4 pærer</td>
</tr>
<tr>
<td>Buy 3 jordbær</td>
</tr>
<tr>
<td>Buy 10 gulerøder</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>shopping.csv</th>
</tr>
</thead>
<tbody>
<tr>
<td>æbler,2</td>
</tr>
<tr>
<td>pærer,4</td>
</tr>
<tr>
<td>jordbær,3</td>
</tr>
<tr>
<td>gulerøder,10</td>
</tr>
</tbody>
</table>
```

CSV file saved with Windows-1252 encoding
JSON

“JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is an ideal data-interchange language.”

www.json.org

- Human readable file format
- Easy way to save a Python expression to a file
- Does not support all Python types, e.g. sets are not supported, and tuples are saved (and later loaded) as lists
JSON example

```python
import json
FILE = 'json-data.json'
data = ((None, True), (42.7, (42,)), [3, 2, 4], (5, 6, 7),
       {'b':'banana', 'a':'apple', 'c': 'coconut'})

with open(FILE, 'w') as outfile:
    json.dump(data, outfile, indent=2, sort_keys=True)

with open(FILE) as infile:
    indata = json.load(infile)

print(indata)
```

```json
[
  [null,
   true
  ],
  [42.7,
   [42
  ]
],
  [3,
   [2,
    4
   ]
  ],
  [5,
   [6,
    7
   ]
  ],
  {
   "a": "apple",
   "b": "banana",
   "c": "coconut"
  }
]
```
XML - eXtensible Markup Language

- XML is a widespread used data format to store hierarchical data with tags and attributes

```xml
cities.xml
<?xml version="1.0"?>
<world>
    <country name="Denmark">
        <city name="Aarhus" pop="264716"/>
        <city name="Copenhagen" pop="1295686"/>
    </country>
    <country name="USA">
        <city name="New York" pop="8622698"/>
        <city name="San Francisco" pop="884363"/>
    </country>
</world>
docs.python.org/3/library/xml.html
```
```python
import xml.etree.ElementTree as ET

FILE = 'cities.xml'

# parse XML file to internal representation
tree = ET.parse(FILE)

# get root element
root = tree.getroot()

for country in root:
    for city in country:
        print(city.attrib['name'], 'in', country.attrib['name'], 'has a population of', city.attrib['pop'])

print(root.tag, root[0][1].attrib) # the tag & indexing the children of an element
print([city.attrib['name'] for city in root.iter('city')]) # .iter finds elements
```

Python shell

Aarhus in Denmark has a population of 264716
Copenhagen in Denmark has a population of 1295686
New York in USA has a population of 8622698
San Francisco in USA has a population of 884363

world {'name': 'Copenhagen', 'pop': '1295686'}
['Aarhus', 'Copenhagen', 'New York', 'San Francisco']
XML tags with text

```xml
<?xml version="1.0"?>
<world>
    <country name="Denmark">
        <city name="Aarhus" pop="264716">The capital of Jutland</city>
        <city name="Copenhagen" pop="1295686">The capital of Denmark</city>
    </country>
    <country name="USA">
        <city name="New York" pop="8622698">Known as Big Apple</city>
        <city name="San Francisco" pop="884363">Home of the Golden Gate Bridge</city>
    </country>
</world>
```

```python
import xml.etree.ElementTree as ET
FILE = 'city-descriptions.xml'
tree = ET.parse(FILE)
root = tree.getroot()

for city in root.iter('city'):
    print(city.get('name'), "-", city.text)
```

<table>
<thead>
<tr>
<th>Aarhus</th>
<th>The capital of Jutland</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copenhagen</td>
<td>The capital of Denmark</td>
</tr>
<tr>
<td>New York</td>
<td>Known as Big Apple</td>
</tr>
<tr>
<td>San Francisco</td>
<td>Home of the Golden Gate Bridge</td>
</tr>
</tbody>
</table>
from openpyxl import Workbook
from openpyxl.styles import Font, PatternFill

wb = Workbook()  # create workbook
ws = wb.active   # active worksheet

ws['A1'] = 42
ws['B3'] = 7
ws['C2'] = ws['A1'].value + ws['B3'].value
ws['D3'] = '=A1+B3+C2'

ws.title = 'My test sheet'

ws['A1'].fill = PatternFill('solid', fgColor='ffff00')
ws['C2'].font = Font(bold=True)

wb.save("openpyxl-example.xlsx")

String searching using `find`

- Search for first occurrence of `substring in str[start, end]`
  
  \[str\].\[find\](\[substring[, start[, end]]\])

- Returns -1 if no occurrence found.

- `.index` similar as `.find`, except raises `ValueError` exception if substring not found

```python
string-search.py

text = 'this is a string - a list of characters'
pattern = 'is'
idx = text.find(pattern)
while idx >= 0:
    print(idx, end= " ")
    idx = text.find(pattern, idx + 1)
```

```
Python shell
|  2  5  22
```
Regular expression
– A powerful language to describe sets of strings

- **Examples**
  - `abc` denotes a string of letters
  - `ab*c` any string starting with `a`, followed by an arbitrary number of `b`s and terminated by `c`, i.e. `{ac, abc, abbc, abbbbc, ...}`
  - `ab+c` equivalent to `abb*c`, i.e. there must be at least one `b`
  - `a\wc` any three letter string, starting with `a` and ending with `c`, where second character is any character in `[a-zA-Z0-9_]`
  - `a[xyz]c` any three letter string, starting with `a` and ending with `c`, where second character is either `x`, `y` or `z`
  - `a[^xyz]c` any three letter string, starting with `a` and ending with `c`, where second character is *none* of `x`, `y` or `z`
  - `^xyz` match at start of string (prefix)
  - `xyz$` match at end of string (suffix)
  - ...

- See [docs.python.org/3/library/re.html](https://docs.python.org/3/library/re.html) for more
String searching using regular expressions

- `re.search(pattern, text)`
  - find the first occurrence of `pattern` in `text` — returns None or a `match object`

- `re.findall(pattern, text)`
  - returns a list of non-overlapping occurrences of `pattern` in `text` — returns a list of substrings

- `re.finditer(pattern, text)`
  - iterator returning a match object for each non-overlapping occurrence of `pattern` in `text`

---

Python shell

```python
import re

text = 'this is a string - a list of characters'
re.findall(r'i\w*', text)  # prefix with 'r' for raw string literal
| ['is', 'is', 'ing', 'ist']

for m in re.finditer(r'a[^at]*t', text):
    print('text[%s, %s] = %s' % (m.start(), m.end(), m.group()))

    text[8, 12] = a st
    text[19, 25] = a list
    text[33, 36] = act
```

[docs.python.org/3/library/re.html](https://docs.python.org/3/library/re.html)
Substitution and splitting using regular expressions

- `re.sub(pattern, replacement, text)`
  - replace any occurrence of the `pattern` in `text` by `replacement`

- `re.split(pattern, text)`
  - split `text` at all occurrences of `pattern`

```python
import re

text = 'this is a string - a list of characters'
re.sub(r'\w*i\w*', 'X', text)  # all words containing i
  # 'X X a X - a X of characters'
re.split(r'[^\w]+a[^\w]+', text)  # split around word 'a'
  # ['this is', 'string', 'list of characters']
```

docs.python.org/3/library/re.html
Assume we want to replace "a" with "an" in front of words starting with the vowels a, e, i, o and u.

---

Python shell

```python
> txt = 'A elephant, a zebra and a ape'
>    # two places to correct
> re.sub('a', 'an', txt)
| 'A elephannt, an zebran annd an anpe'
>    # replaces all letters 'a' with 'an'
> re.sub(r'\ba\b', 'an', txt)
| 'A elephant, an zebra and an ape'
>    # raw string + \b boundary of word
> re.sub(r'\b[aA]\b', 'an', txt)
| 'an elephant, an zebra and an ape'
>    # all lower 'a' replaced
> re.sub(r'\b([aA])\b', r'\1\2', txt)
| 'An elephant, a zebra and an ape'
>    # use () and \1 to reinsert match
> re.sub(r'\b([aA])\s+[aeiou]\b', r'\1\2', txt)
| 'Anelephant, a zebra and anpe'
>    # missing original whitespace + vowel
> re.sub(r'\b([aA])\s+[aeiou]\b', r'\1\2', txt)
| 'An elephant, a zebra and an ape'
```
Fun with strings: Lindenmayer systems (L-systems)

**Axiom**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ABA</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ABAAB</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ABAABABA</td>
<td></td>
</tr>
</tbody>
</table>

**First four iterations of parallel rewriting**

```
A
AB
ABA
ABAAB
```

**Rules**

A → AB
B → A

```
L_system.py

S = 'A'  # axiom
rules = {'A': 'AB', 'B': 'A'}
for i in range(8):
    S = ''.join(rules.get(c, c) for c in S)
    print(S)
```

**Python shell**

```
| AB |
| ABA |
| ABAAB |
| ABAABABA |
| ABAABABAABAABAABABAABABAABAABABAABABA |
| ABAABABAABAABAABABAABABAABABAABAABABAABABAABAABABA |
```

“L-systems were introduced and developed in 1968 by Aristid Lindenmayer, a Hungarian theoretical biologist and botanist at the University of Utrecht. Lindenmayer used L-systems to describe the behaviour of plant cells and to model the growth processes of plant development.”

- Wikipedia

[en.wikipedia.org/wiki/L-system](en.wikipedia.org/wiki/L-system)
Heighway Dragon

dragon.py

```python
import matplotlib.pyplot as plt
from math import sin, cos, radians

axiom = 'FX'
rules = {'X': 'X+YF+', 'Y': '-FX-Y'}

def apply_rules(axiom, rules, repeat):
    for _ in range(repeat):
        axiom = ''.join(rules.get(symbol, symbol) for symbol in axiom)
    return axiom

def walk(commands, position=(0, 0), angle=0, turn=90):
    path = [position]
    for move in commands:
        if move == 'F':
            position = (position[0] + cos(radians(angle)),
                                position[1] + sin(radians(angle)))
            path.append(position)
        elif move == '-': angle -= turn
        elif move == '+': angle += turn
    return path

path = walk(apply_rules(axiom, rules, 13))
plt.plot(*zip(*path), '-')
plt.title('Heighway dragon')
plt.show()
```

Interprete the symbols of the resulting string as a walk where 'F' = draw line forward, and '+' and '-' are turn left and right 90° (X and Y are skipped)
More space filling curves...

Sierpinski triangle
Axiom F-G-G
F → F-G+F+G-F
G → GG
Forward F and G
Turns 120°

Heighway dragon
Axiom FX
X → X+YF+
Y → -FX-Y

McWorter Pentigree curve
Axiom F-F-F-F-F
F → F-F-F++F+F-F
Turns 72°

Sierpinski arrowhead curve
Axiom A
A → B-A-B
B → A+B+A
Forward A and B
Turns 60°

Koch curve
Axiom F
F → F+F-F+F+F

Peano curve
Axiom L
L → LFRFL-F-RFLFR+F+LFRFL
R → RFLFR+F+LFRFL-F-RFLFR

Hilbert curve
Axiom L
L → +RF-LFL-FR+
R → -LF+RFR+FL-

Tree
Axiom F
F → F[+FF][FF]F[-F][F][F+F]
tTurns 36°
[
and
] return to start point when done

Cesero fractal
Axiom F
F → F+F--F+F
Turns 80°
import matplotlib.pyplot as plt
from math import sin, cos, radians

def walk(commands, pos=(0, 0), forward=frozenset('F'), angle=0, turn=90):
    paths = [[pos]]
    stack = []
    for move in commands:
        if move in forward:
            pos = (pos[0] + cos(radians(angle)), pos[1] + sin(radians(angle)))
            paths[-1].append(pos)
        elif move == '-': angle -= turn
        elif move == '+': angle += turn
        elif move == '[':
            stack.append((pos, angle))
        elif move == ']':
            pos, angle = stack.pop()
            paths.append([pos])
    return paths

def apply_rules(axiom, rules, repeat=1):
    for _ in range(repeat):
        axiom = ''.join(rules.get(symbol, symbol) for symbol in axiom)
    return axiom

curves = [
    ('Sierpinski triangle', 'F-G-G', {'F': 'F-G+F-G-F', 'G': 'GG'}, 5, {'turn': 120, 'forward': {'F', 'G'}}),
    ('Sierpinski arrowhead curve', 'A', {'A': 'B-A-B', 'B': 'B+A+B'}, 5, {'turn': 60, 'forward': {'A', 'B'}}),
    ('Peano curve', 'L', {'L': 'LFRFL-FRFLFR+PRFRFL', 'R': 'RFLFR+PFRFR+PFRFL'}, 3,{}),
    ('Heighway dragon', 'FX', {'X': 'X+YF+', 'Y': '-FX-Y'}, 10,{}),
    ('Koch curve', 'F', {'F': 'F+F-F+F'}, 3,{}),
    ('Hilbert curve', 'L', {'L': '+RF-LFL-+FR', 'R': '+RF-+FRF-L'}, 3,{}),
    ('McWorter Pentigree curve', 'F-F-F-F', {'F': 'F-F-F+F+F-F-F'}, 3,{'turn': 72}),
    ('Tree', 'F', {'F': 'F[+FR][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+FF][+F

for idx, (title, axiom, rules, repeat, walk_arg) in enumerate(curves, start=1):
    paths = walk(apply_rules(axiom, rules, repeat), **walk_arg)
    ax = plt.subplot(3, 3, idx, aspect='equal')
    ax.set_title(title)
    for path in paths:
        plt.plot(*zip(*path), '-' )
    plt.axis('off')
plt.show()
Relational data

- SQLite
- pandas
## Table: country

<table>
<thead>
<tr>
<th>name</th>
<th>population</th>
<th>area</th>
<th>capital</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Denmark'</td>
<td>5748769</td>
<td>42931</td>
<td>'Copenhagen'</td>
</tr>
<tr>
<td>'Germany'</td>
<td>82800000</td>
<td>357168</td>
<td>'Berlin'</td>
</tr>
<tr>
<td>'USA'</td>
<td>325719178</td>
<td>9833520</td>
<td>'Washington, D.C.'</td>
</tr>
<tr>
<td>'Iceland'</td>
<td>334252</td>
<td>102775</td>
<td>'Reykjavik'</td>
</tr>
</tbody>
</table>

## Table: city

<table>
<thead>
<tr>
<th>name</th>
<th>country</th>
<th>population</th>
<th>established</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Copenhagen'</td>
<td>'Denmark'</td>
<td>775033</td>
<td>800</td>
</tr>
<tr>
<td>'Aarhus'</td>
<td>'Denmark'</td>
<td>273077</td>
<td>750</td>
</tr>
<tr>
<td>'Berlin'</td>
<td>'Germany'</td>
<td>3711930</td>
<td>1237</td>
</tr>
<tr>
<td>'Munich'</td>
<td>'Germany'</td>
<td>1464301</td>
<td>1158</td>
</tr>
<tr>
<td>'Reykjavik'</td>
<td>'Iceland'</td>
<td>126100</td>
<td>874</td>
</tr>
<tr>
<td>'Washington D.C.'</td>
<td>'USA'</td>
<td>693972</td>
<td>1790</td>
</tr>
<tr>
<td>'New Orleans'</td>
<td>'USA'</td>
<td>343829</td>
<td>1718</td>
</tr>
<tr>
<td>'San Francisco'</td>
<td>'USA'</td>
<td>884363</td>
<td>1776</td>
</tr>
</tbody>
</table>
SQL pronounced ˌɛsˌkjjuːˈɛl or ˈsiːkwəl

- **SQL** = Structured Query **Language**
- **Database** = collection of tables stored persistently on disk
- ANSI and ISO standards since 1986 and 1987, respectively; origin early 70s
  
- Widespread used SQL databases (can handle many tables/rows/users): Oracle, MySQL, Microsoft SQL Server, PostgreSQL and IBM DB2

- **SQLite** is a very lightweight version storing a database in a single file, without a separate database server

- SQLite is included in both iOS and Android mobil phones

### Table: country

<table>
<thead>
<tr>
<th>name</th>
<th>population</th>
<th>area</th>
<th>capital</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Denmark'</td>
<td>5748769</td>
<td>42931</td>
<td>'Copenhagen'</td>
</tr>
<tr>
<td>'Germany'</td>
<td>82800000</td>
<td>357168</td>
<td>'Berlin'</td>
</tr>
<tr>
<td>'USA'</td>
<td>325719178</td>
<td>9833520</td>
<td>'Washington, D.C.'</td>
</tr>
<tr>
<td>'Iceland'</td>
<td>334252</td>
<td>102775</td>
<td>'Reykjavik'</td>
</tr>
</tbody>
</table>

The Course “Database Systems” gives a more in-depth introduction to SQL (MySQL)
CREATE TABLE country (name, population, area, capital)

INSERT INTO country VALUES ('Denmark', 5748769, 42931, 'Copenhagen')

UPDATE country SET population=5748770 WHERE name='Denmark'

SELECT name, capital FROM country WHERE population >= 1000000

> [('Denmark', 'Copenhagen'), ('Germany', 'Berlin'), ('USA', 'Washington, D.C.')] 

SELECT * FROM country WHERE capital = 'Berlin'

> [('Germany', 82800000, 357168, 'Berlin')] 

SELECT country.name, city.name, city.established FROM city, country
WHERE city.name=country.capital AND city.population < 500000

> [('Iceland', 'Reykjavik', 874), ('USA', 'Washington, D.C.', 1790)]

DELETE FROM country WHERE name = 'Germany'

DROP TABLE country

Table: country

<table>
<thead>
<tr>
<th>name</th>
<th>population</th>
<th>area</th>
<th>capital</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Denmark'</td>
<td>5748769</td>
<td>42931</td>
<td>'Copenhagen'</td>
</tr>
<tr>
<td>'Germany'</td>
<td>82800000</td>
<td>357168</td>
<td>'Berlin'</td>
</tr>
<tr>
<td>'USA'</td>
<td>325719178</td>
<td>9833520</td>
<td>'Washington, D.C.'</td>
</tr>
<tr>
<td>'Iceland'</td>
<td>334252</td>
<td>102775</td>
<td>'Reykjavik'</td>
</tr>
</tbody>
</table>
import sqlite3

c = connection.cursor()
c.executescript('DROP TABLE IF EXISTS country; DROP TABLE IF EXISTS city;')

countries = [('Denmark', 5748769, 42931, 'Copenhagen'),
             ('Germany', 82800000, 357168, 'Berlin'),
             ('USA', 325719178, 9833520, 'Washington, D.C.'),
             ('Iceland', 334252, 102775, 'Reykjavik')]

cities = [('Copenhagen', 'Denmark', 775033, 800),
          ('Aarhus', 'Denmark', 273077, 750),
          ('Berlin', 'Germany', 3711930, 1237),
          ('Munich', 'Germany', 1464301, 1158),
          ('Reykjavik', 'Iceland', 126100, 874),
          ('Washington, D.C.', 'USA', 693972, 1790),
          ('New Orleans', 'USA', 343829, 1718),
          ('San Francisco', 'USA', 884363, 1776)]

c.execute('CREATE TABLE country (name, population, area, capital)')
c.execute('CREATE TABLE city (name, population, established)')
c.executemany('INSERT INTO country VALUES (?, ?, ?, ?)', countries)
c.executemany('INSERT INTO city VALUES (?, ?, ?, ?)', cities)

c.commit()  # save data to database before closing

c.close()
### SQLite query examples

#### sqlite-example.py

```python
for row in c.execute('SELECT * FROM country'):  # * = all columns, execute returns iterator
    print(row)  # row is by default a Python tuple

for row in c.execute('SELECT * FROM city, country -- all pairs of rows from city × country
                        WHERE city.name = country.capital AND city.population < 700000'):  
    print(row)

print(*c.execute('''SELECT country.name,  
                   COUNT(city.name) AS cities,  
                   100 * SUM(city.population) / country.population  
                   FROM city JOIN country ON city.country = country.name  -- SQL join 2 tables
                   WHERE city.population > 500000  -- only consider big cities
                   GROUP BY city.country  -- output has one row per group of rows
                   ORDER BY cities DESC, SUM(city.population) DESC'''))  # ordering of output
```

#### Python shell

```
| ('Denmark', 5748769, 42931, 'Copenhagen')  
| ('Germany', 82800000, 357168, 'Berlin')  
| ('USA', 325719178, 9833520, 'Washington, D.C.')  
| ('Iceland', 334252, 102775, 'Reykjavik')  
| ('Reykjavik', 'Iceland', 126100, 874, 'Iceland', 334252, 102775, 'Reykjavik')  
| ('Washington, D.C.', 'USA', 693972, 1790, 'USA', 325719178, 9833520, 'Washington, D.C.')  
| ('Germany', 2, 6) ('USA', 2, 0) ('Denmark', 1, 13)
```
```python
import sqlite3

connection = sqlite3.connect('users.sqlite')
c = connection.cursor()
c.execute('CREATE TABLE users (name)')

while True:
    user = input('New user: ')
    c.executescript('INSERT INTO users VALUES ("%s")' % user)
    connection.commit()
    print(list(c.execute('SELECT * FROM users')))
```

Insecure: NEVER use `%` on user input

Right way:
```
c.execute('INSERT INTO users VALUES (?)', (user,))
```
HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.

OH, DEAR - DID HE BREAK SOMETHING? IN A WAY-

DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;--?

OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.

WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.

AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.
Pandas

- Comprehensive Python library for data manipulation and analysis, in particular tables and time series
- Pandas **data frames** = tables
- Supports interaction with SQL, CSV, JSON, ...
- Integrates with Jupyter, numpy, matplotlib, ...

\[ y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it} \]
Pandas integration with Jupyter

- Tables (Pandas data frames) are rendered nicely in Jupyter

```python
In [1]:
1import pandas as pd
2students = pd.read_csv('students.csv')
3students

Out[1]:
```
```
<table>
<thead>
<tr>
<th>Name</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Donald Duck</td>
<td>Copenhagen</td>
</tr>
<tr>
<td>Goofy</td>
<td>Aarhus</td>
</tr>
<tr>
<td>Mickey Mouse</td>
<td>Aarhus</td>
</tr>
</tbody>
</table>
```

students.csv
Name,City
"Donald Duck","Copenhagen"
"Goofy","Aarhus"
"Mickey Mouse","Aarhus"
Reading tables (data frames)

- Pandas provide functions for reading different data formats, e.g. SQLite and .csv files, into pandas.DataFrames.

```python
import pandas as pd
import sqlite3
connection = sqlite3.connect('example.sqlite')
countries = pd.read_sql_query('SELECT * FROM country', connection)
cities = pd.read_sql_query('SELECT * FROM city', connection)
students.to_sql('students', connection, if_exists='replace')
print(students)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Donald Duck</td>
<td>Copenhagen</td>
</tr>
<tr>
<td>Goofy</td>
<td>Aarhus</td>
</tr>
<tr>
<td>Mickey Mouse</td>
<td>Aarhus</td>
</tr>
</tbody>
</table>
Selecting columns and rows

Python shell

```python
> countries['name'] # select column
> countries.name # same as above
> countries[['name', 'capital']] # select multiple columns, note double-[]
> countries.head(2) # first 2 rows
> countries[1:3] # slicing rows, rows 1 and 2
> countries[::2] # slicing rows, rows 0 and 2
> countries.at[1, 'area'] # indexing cell by (row label, column name)
> cities[(cities['name'] == 'Berlin') | (cities['name'] == 'Munich')] # select rows

<table>
<thead>
<tr>
<th>name</th>
<th>country</th>
<th>population</th>
<th>area</th>
<th>capital</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berlin</td>
<td>Germany</td>
<td>3711930</td>
<td>1237</td>
<td>'Copenhagen'</td>
</tr>
<tr>
<td>Munich</td>
<td>Germany</td>
<td>1464301</td>
<td>1158</td>
<td></td>
</tr>
</tbody>
</table>

> pd.DataFrame([[1,2], [3,4], [5,6]], columns=['x', 'y']) # create DF from list
> pd.DataFrame(np.random.random((3,2)), columns=['x', 'y']) # from numpy
```

Table: country

<table>
<thead>
<tr>
<th>name</th>
<th>population</th>
<th>area</th>
<th>capital</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Denmark'</td>
<td>5748769</td>
<td>42931</td>
<td>'Copenhagen'</td>
</tr>
<tr>
<td>'Germany'</td>
<td>82800000</td>
<td>357168</td>
<td>'Berlin'</td>
</tr>
<tr>
<td>'USA'</td>
<td>325719178</td>
<td>9833520</td>
<td>'Washington, D.C.'</td>
</tr>
<tr>
<td>'Iceland'</td>
<td>334252</td>
<td>102775</td>
<td>'Reykjavik'</td>
</tr>
</tbody>
</table>
```python
Python shell

```
Merging tables and creating a new column

```python
M = pd.merge(countries, cities, left_on='capital', right_on='name')
# both data frames had a 'name' and 'population' column
M1 = M.rename(columns={
    'population_x': 'country_population',
    'population_y': 'capital_population'
})
M2 = M1.drop(columns=['name_x', 'name_y'])
M2['%pop in capital'] = M2.capital_population / M2.country_population
M2.sort_values('%pop in capital', ascending=False, inplace=True)
print(M2[['country', '%pop in capital']])
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Iceland</td>
<td>0.377260  # note row labels are permuted</td>
</tr>
<tr>
<td>0</td>
<td>Denmark</td>
<td>0.134817</td>
</tr>
<tr>
<td>1</td>
<td>Germany</td>
<td>0.044830</td>
</tr>
<tr>
<td>2</td>
<td>USA</td>
<td>0.002131</td>
</tr>
</tbody>
</table>
```
Pandas datareader and Matplotlib

- pandas_datareader provides access to many data sources
- dataframes have a .plot method (using matplotlib.pyplot)

```python
import matplotlib.pyplot as plt
import pandas_datareader

#df = pandas_datareader.data.DataReader(['AAPL', 'GOOGL', 'MSFT', 'ZM'], 'stooq')  # ignores start=...

df = pandas_datareader.stooq.StooqDailyReader(['AAPL', 'GOOGL', 'MSFT', 'ZM'], start='2000-01-01').read()

df['Close'].plot()
plt.legend()
plt.show()
```

- pandas_datareader provides access to many data sources
- dataframes have a .plot method (using matplotlib.pyplot)
Hierarchical / Multi-level indexing (MultiIndex)

Python shell
```
> df.tail(2)
<table>
<thead>
<tr>
<th>Attributes</th>
<th>Close</th>
<th>...</th>
<th>Volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbols</td>
<td>AAPL</td>
<td>GOOGL</td>
<td>MSFT</td>
</tr>
<tr>
<td>Date</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2020-04-29</td>
<td>287.73</td>
<td>1342.18</td>
<td>177.43</td>
</tr>
<tr>
<td>2020-04-30</td>
<td>293.80</td>
<td>1346.70</td>
<td>179.21</td>
</tr>
</tbody>
</table>
```

Both rows and columns can have multi-level indexing

Python shell
```
> df['Close'].tail(2)
<table>
<thead>
<tr>
<th>Symbols</th>
<th>AAPL</th>
<th>GOOGL</th>
<th>MSFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2020-04-29</td>
<td>1342.18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2020-04-30</td>
<td>1346.70</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
> df['Close']['GOOGL'].tail(2)
<table>
<thead>
<tr>
<th>Date</th>
<th>2020-04-29</th>
<th>2020-04-30</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1342.18</td>
<td>1346.70</td>
</tr>
</tbody>
</table>
```

```
> df.loc[:, pd.IndexSlice[:, 'GOOGL']].tail(2)
<table>
<thead>
<tr>
<th>Attributes</th>
<th>Close</th>
<th>High</th>
<th>Low</th>
<th>Open</th>
<th>Volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbols</td>
<td>GOOGL</td>
<td>GOOGL</td>
<td>GOOGL</td>
<td>GOOGL</td>
<td>GOOGL</td>
</tr>
<tr>
<td>Date</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2020-04-29</td>
<td>1342.18</td>
<td>1360.15</td>
<td>1326.73</td>
<td>1345.00</td>
<td>5417888.0</td>
</tr>
<tr>
<td>2020-04-30</td>
<td>1346.70</td>
<td>1350.00</td>
<td>1321.50</td>
<td>1331.36</td>
<td>2788644.0</td>
</tr>
</tbody>
</table>
```

[pandas.pydata.org/pandas-docs/stable/user_guide/advanced.html](pandas.pydata.org/pandas-docs/stable/user_guide/advanced.html)
Clustering

- k-means
- scipy.cluster.vq.kmeans
- neural networks
3 clusters / groups of points
Clustering = Optimization problem

Example: k-means

- Find $k$ points *centroids*
- Assign each input point to nearest centroid $\rightarrow$ $k$ clusters $C$
- \[ \text{distortion} = \sum_{C \in C} \sum_{p \in C} |p - \text{centroid}(C)|^2 \]
- **Goal**: Find $k$ centroids that minimize distortion
k-means for $k = 1$

- Let the centroid point $c$ for a point set $C$ be the point minimizing the distortion:

$$\text{distortion} = \sum_{p \in C} |p - c|^2$$

- Theorem: $c = \text{average}(C)$
k-means - Lloyd’s method (pseudo code)

centroids = k distinct random input points
while centroids change:
    create clusters C by assigning points to the nearest centroid
    centroids = average of each cluster
k-means is a heuristic. Output can be far from optimal.
Generating random points (just one random approach)

```python
from random import random
from math import pi, cos, sin

def random_point(x, y, radius):
    angle = 2 * pi * random()
    r = radius * random() ** 2
    return x + r * cos(angle), y + r * sin(angle)

def random_points(n, x, y, radius):
    for _ in range(n):
        yield random_point(x, y, radius)
```

```python
from random import sample
from numpy import argmin, mean

def k_means(points, k):
    centroid = sample(points, k)
    centroids = [centoid]  # history for visualization
    while True:
        clusters = [[] for _ in centroid]
        for p in points:
            i = argmin([dist(p, c) for c in centroid])
            clusters[i].append(p)
        centroid = [tuple(map(mean, zip(*c))) for c in clusters]
        if centroid == centroids[-1]:
            break
        centroids.append(centroid)
        if min(len(c) for c in clusters) == 0:
            print("Not good - empty cluster")
            break
    return clusters
```

k-mean limitations

- Can easily converge to a solution **far from a global minimum**
  - Solution – try several times and take the best
    (possibly since we can measure the quality (= distortion) of a solution)

- **Clusters can become empty**
  - Solution – discard and restart / take a random point out as a new centroid /
    take point furthest away from existing centroids / ....

- **Sensitive to the scales of the different dimensions**
  - Solution – apply some kind of initial normalization of coordinates
k-means - better bounds

- The *k*-means++ algorithm achieves an expected guarantee to be at most a factor $8(\ln k + 2)$ from the optimal [Vassilvitskii & Arthur]

- There exist polynomial time approximation schemes that find a solution that is guaranteed $1 + \varepsilon$ of the optimal (but running time exponential in $k$ and dimension of points) [Har Peled et al.]

- In practice: A heuristic is most often the algorithm of choice
scipy.cluster.vq.kmeans

k_means.py

```python
from scipy.cluster.vq import kmeans, whiten
import matplotlib.pyplot as plt

points = whiten(points)  # normalize variance of points

plt.plot(*zip(*points), 'r.')
plt.plot(*zip(*kmeans(points, K)[0]), "bo")
plt.title("scipy.cluster.vq.kmeans")
```

**Note:** According to the documentation "whiten must be called prior to passing an observation matrix to kmeans"

kmeans returns tuple (centroids, distortion)

docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.vq.kmeans.html
scipy.cluster.vq.whiten

- Normalizes / scales each dimension to have unit variance 1.0

\[
\text{Var}(X) = \frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2 \\
\mu = \frac{1}{n} \sum_{i=1}^{n} x_i
\]

Other Python clustering methods - `sklearn.cluster`

Data Mining Algorithms

- k-means, and more generally clustering, is just one field in the area of *Data Mining*

- For more information see the webpage [Top 10 Data Mining Algorithms, Explained](www.kdnuggets.com/2015/05/top-10-data-mining-algorithms-explained.html), a follow up to the below paper

Neural networks (one slide introduction)

MNIST : 28 x 28 pixel values from [0, 255]

Classification, like MNIST, prediction = index of node with maximum output

Common activation functions

Learning
Find As and bs performing well (minimize a cost function) on a set of n training inputs x with known output y using backpropagation / stochastic gradient descend
Applying a linear classifier using Numpy: \( x \cdot A + b \)

```python
import matplotlib.pyplot as plt
import numpy as np
import keras

(train_images, train_labels), (test_images, test_labels) = keras.datasets.mnist.load_data()
type(test_images) > <class 'numpy.ndarray'>
test_images.shape > (10000, 28, 28) # 10_000 images 28 x 28
test_labels.shape > (10000,) # 10_000 labels
test_labels[:3] > array([7, 2, 1], dtype=uint8)

for i, image in zip(range(3), test_images):
    plt.subplot(1, 3, i + 1)
    plt.imshow(image)
plt.show()

A, b = map(np.array, eval(open('mnist_linear.weights').read())) # read A and b from file
print(A.shape, A.dtype, b.shape, b.dtype)
> (784, 10) float64 (10,) float64

print([np.argmax(image.reshape(28 * 28) @ A + b) for image in test_images[:3]])
> [7, 2, 1] # correct on 9_142 of the 10_000 images for the above file, ie accuracy 91%
```
THIS IS YOUR MACHINE LEARNING SYSTEM?

YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.

WHAT IF THE ANSWERS ARE WRONG?

JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
Graphical user interfaces (GUI)

- Tkinter
accumulator = 0

while True:
    print("Accumulator:", accumulator)
    print("Select:")
    print("  1: clear")
    print("  2: add")
    print("  3: subtract")
    print("  4: multiply")
    print("  5: quit")

    choice = int(input("Choice: "))

    if choice == 1: accumulator = 0
    if choice == 2: accumulator += int(input("add: "))
    if choice == 3: accumulator -= int(input("subtract: "))
    if choice == 4: accumulator *= int(input("multiply by: "))
    if choice == 5: break
Python GUI’s (Graphical Users Interfaces)

- There is a long list of GUI frameworks and toolkits, designer tools – we will only briefly look at Tkinter
- GUI are, opposed to a text terminal, easier to use, more intuitive and flexible
- Windows, icons, menus, buttons, scrollbars, mouse / touch / keyboard interaction etc.
- Operating system (e.g. Windows, macOS, iOS, Linux, Android) provides basic functionality in particular a windows manager
- Writing GUI applications from scratch can be painful – frameworks try to provide all standard functionality

[link](wiki.python.org/moin/GuiProgramming)
[link](en.wikipedia.org/wiki/Colossal_Cave_Adventure)
Tkinter

- “Tkinter is Python's de-facto standard GUI (Graphical User Interface) package. It is a thin object-oriented layer on top of Tcl/Tk.”
- “Tcl is a high-level, general-purpose, interpreted, dynamic programming language.”
- “Tk is a free and open-source, cross-platform widget toolkit that provides a library of basic elements of GUI widgets for building a graphical user interface (GUI) in many programming languages.”
- “The popular combination of Tcl with the Tk extension is referred to as Tcl/Tk, and enables building a graphical user interface (GUI) natively in Tcl. Tcl/Tk is included in the standard Python installation in the form of Tkinter.”
Terminology

- **widgets** (e.g. buttons, editable text fields, labels, scrollbars, menus, radio buttons, check buttons, canvas for drawing, frames...)
- **events** (e.g. mouse click, mouse entering/leaving, resizing windows, redraw requests, ...)
- **listening** (application waits for events to be fired)
- **event handler** (a function whose purpose is to handle an event, many triggered by user or OS/Window manager)
- **geometry managers** (how to organize widgets in a window: Tkinter `pack`, `grid`, `place`)
Graphical user interfaces (GUI)

- Tkinter
“tkinter is also famous for having an outdated look and feel”
Welcome example

welcome.py

```python
import tkinter

root = tkinter.Tk()  # root window

def do_quit():  # event handler for "Close" button
    root.destroy()

root.title("Tkinter Welcome GUI")

label = tkinter.Label(root, text="Welcome to Tkinter", background="yellow",
                       anchor=tkinter.SE, font=("Helvetica", "24", "bold italic"),
                       padx=10, pady=10)

label.pack(side=tkinter.LEFT, fill=tkinter.BOTH, expand=True)

close_button = tkinter.Button(root, text="Close", command=do_quit)

close_button.pack(side=tkinter.RIGHT)

tkinter.mainloop()  # loop until all windows are closed/destroyed
```
import tkinter

class Welcome:
    def do_quit(self):  # event handler for "Close"
        self.root.destroy()

    def __init__(self, window_title):
        self.root = tkinter.Tk()
        self.root.title(window_title)

        self.label = tkinter.Label(self.root, text="Welcome")
        self.label.pack(side=tkinter.LEFT)

        self.close_button = tkinter.Button(self.root, text="Close", command=self.do_quit)
        self.close_button.pack(side=tkinter.RIGHT)

Welcome("My Window")

tkinter.mainloop()
import tkinter

class Counter:
    def do_quit(self):
        self.root.destroy()

    def add(self, x):
        self.counter += x
        self.count.set(self.counter)

    def __init__(self, message):
        self.counter = 0
        self.root = tkinter.Toplevel()  # new window
        self.root.title("Counter")
        self.label = tkinter.Label(self.root, text=message)
        self.label.grid(row=0, columnspan=3)
        self.minus_button = tkinter.Button(self.root, text="-", command=lambda: self.add(-1))
        self.minus_button.grid(row=1, column=0)
        self.count = tkinter.IntVar()
        self.count_label = tkinter.Label(self.root, textvariable=self.count)
        self.count_label.grid(row=1, column=1)
        self.plus_button = tkinter.Button(self.root, text="+", command=lambda: self.add(+1))
        self.plus_button.grid(row=1, column=2)
import tkinter

class Counter_app:
    def __init__(self):
        self.counters = 0
        self.root = tkinter.Tk()
        self.create = tkinter.Button(self.root, text="Create counter", command=self.new_counter)
        self.create.pack()

    def new_counter(self):
        Counter("Counter " + chr(ord('A') + self.counters))
        self.counters += 1

Counter_app()
tkinter.mainloop()
import tkinter

root = tkinter.Tk()

canvas = tkinter.Canvas(root, width=100, height=100)
canvas.pack()
canvas.create_line(0, 0, 100, 100)
canvas.create_oval(20, 20, 80, 80, fill="blue")

close = tkinter.Button(root, text="Close", command=root.destroy)
close.pack()

tkinter.mainloop()
import tkinter
from tkinter import messagebox

class Calculator:
    def __init__(self, root):
        self.root = root
        self.display = tkinter.Entry(self.root, font=('Helvetica', 16), justify=tkinter.RIGHT)
        self.display.insert(0, '0')
        self.display.grid(row=0, column=0, columnspan=5)  # grid = geometry manager

        self.button(1, 0, '7')
        self.button(1, 1, '8')
        self.button(1, 2, '9')
        self.button(1, 3, '*')
        self.button(1, 4, 'C', command=self.clearText)  # 'C' button

        self.button(2, 0, '4')
        self.button(2, 1, '5')
        self.button(2, 2, '6')
        self.button(2, 3, '/')
        self.button(2, 4, '%')

        self.button(3, 0, '1')
        self.button(3, 1, '2')
        self.button(3, 2, '3')
        self.button(3, 3, '-')

        self.button(3, 4, '=', rowspan=2, command=self.calculateExpression)  # '=' button

        self.button(4, 0, '0', colspan=2)
        self.button(4, 2, '.')
        self.button(4, 3, '+')
def button(self, row, column, text, command=None, columnspan=1, rowspan=1):
    if command == None:
        command = lambda: self.appendToDisplay(text)
    B = tkinter.Button(self.root, font=('Helvetica', 11), text=text, command=command)
    B.grid(row=row, column=column, rowspan=rowspan, columnspan=columnspan, sticky="NWNESWSE")

def clearText(self):
    self.replaceText("0")

def replaceText(self, text):
    self.display.delete(0, tkinter.END)
    self.display.insert(0, text)

def appendToDisplay(self, text):
    if self.display.get() == "0":
        self.replaceText(text)
    else:
        self.display.insert(tkinter.END, text)

def calculateExpression(self):
    expression = self.display.get().replace("%", "/ 100")
    try:
        result = eval(expression)  # DON'T DO THIS !!!
        self.replaceText(result)
    except:
        messagebox.showinfo("Message", "Invalid expression", icon="warning")

root = tkinter.Tk()
root.title("Calculator")
root.resizable(0, 0)
Calculator(root)
tkinter.mainloop()
Creating a menu

class Rectangles:
    Colors = ['black', 'red', 'blue', 'green', 'yellow']

    def create_menu(self):
        menubar = tkinter.Menu(self.root)
        menubar.add_command(label="Quit! (Ctrl-q)", command=self.do_quit)

        editmenu = tkinter.Menu(menubar, tearoff=0)
        editmenu.add_command(label="Clear", command=self.clear_all)
        editmenu.add_command(label="Delete last (Ctrl-z)", command=self.delete_last_rectangle)

        colormenu = tkinter.Menu(menubar, tearoff=0)
        for color in self.Colors:
            colormenu.add_command(label=color,
                                foreground=color,
                                command=self.get_color_handler(color))

        menubar.add_cascade(label="Edit", menu=editmenu)
        menubar.add_cascade(label="Color", menu=colormenu)
        self.root.config(menu=menubar)  # Show menubar

    def get_color_handler(self, color):
        return lambda : self.set_color(color)

    def set_color(self, color):
        self.current_color = color

...
Binding key and mouse events

- Whenever a key is pressed, mouse button is pressed/released, mouse is moved, mouse enters/leaves objects etc. events are triggered that can be bound to call a user defined event handler

```
rectangles.py  (continued)
...
self.root = tkinter.Tk()
self.root.bind('<Control-q>', self.do_quit)
self.root.bind('<Control-z>', self.delete_last_rectangle)
...
self.canvas = tkinter.Canvas(self.root, width=300, height=200, background='white')
self.canvas.bind('<Button-1>', self.create_rectangle_start)
self.canvas.bind('<B1-Motion>', self.create_rectangle_mouse_move)
self.canvas.bind('<ButtonRelease-1>', self.create_rectangle_end)
...```
def create_rectangle_start(self, event):
    radius = 3
    x, y = event.x, event.y
    self.top_pos = (x, y)
    self.bottom_pos = (x, y)
    self.rectangle = self.canvas.create_rectangle(x, y, x, y,  # top-left = bottom-right
                                                fill=self.current_color, width=1, outline='grey', dash=(3, 5))
    self.corner = self.canvas.create_oval(x - radius, y - radius, x + radius, y + radius, fill='white')

def create_rectangle_mouse_move(self, event):
    if self.corner is not None:
        x, y = event.x, event.y
        x_, y_ = self.bottom_pos
        self.bottom_pos = (x, y)
        self.canvas.move(self.corner, x - x_, y - y_)

def create_rectangle_end(self, event):
    if self.corner is not None:
        self.canvas.delete(self.corner)
        self.corner = None
        if self.bottom_pos != self.top_pos:
            self.rectangles.append(self.rectangle)
            self.canvas.itemconfig(self.rectangle, width=0)
        else:
            # empty rectangle, skip
            self.canvas.delete(self.rectangle)
            self.rectangle = None
Exercise 25.1 (convex hull GUI)