How to make code beautiful

Python beyond basics

2019-04-03 Session 2 Peter Regner

How to make code beautiful

Disclaimers:

- based on experience, not the full truth
- Python 3! forget Python 2 (except some exotic libraries)
- no Windows questions please, Linux only ;)

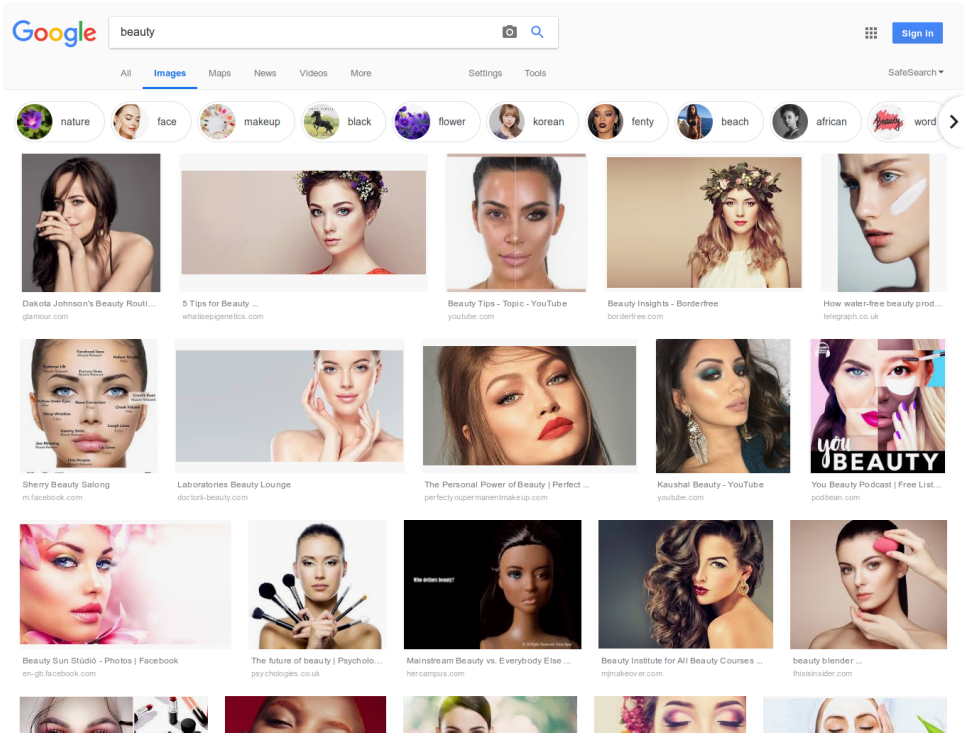
How to make code beautiful

Goals:

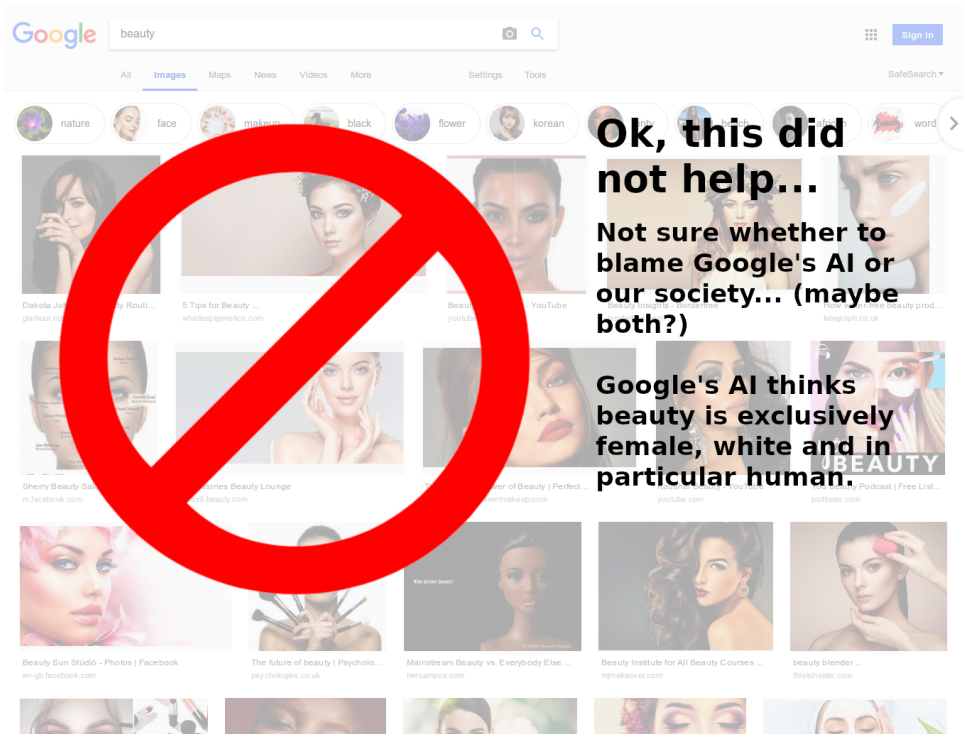
- a bit of general blabla about programming
- point at specific features and gotchas of Python
- some important tips, tricks and tools

Questions?

What is Beautiful Code?



What is Beautiful Code?



BTW Google's AI puts "nature" as first keyword, but the third "makeup" seems to be more relevant for selecting the pictures shown by default.

What is Beautiful Code?

> **Beauty** is a [...] characteristic of [...] idea [...], that provides a perceptual experience of **pleasure** or **satisfaction**.

> [...]
>
> An "ideal beauty" is an entity which is [...] possesses **features** widely attributed to beauty **in a particular culture**, for perfection.

Source: https://en.wikipedia.org/wiki/Beauty

What is Beautiful Code?

In [1]:
```
import this
```

```
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!
```

What is Beautiful Code?

In [3]:
```
import antigravity
```

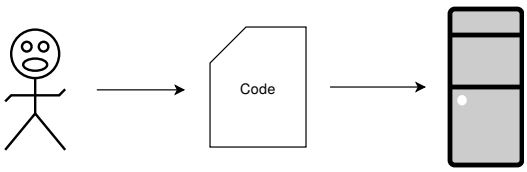
What is Beautiful Code?

In the Python universe:

beauty = simplicity

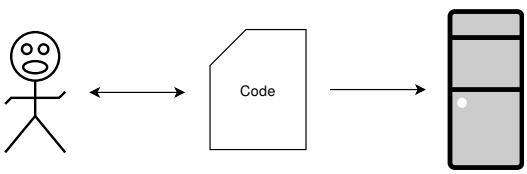
in terms of clarity, easy to comprehend, easy to write, not complicated

What is Code?

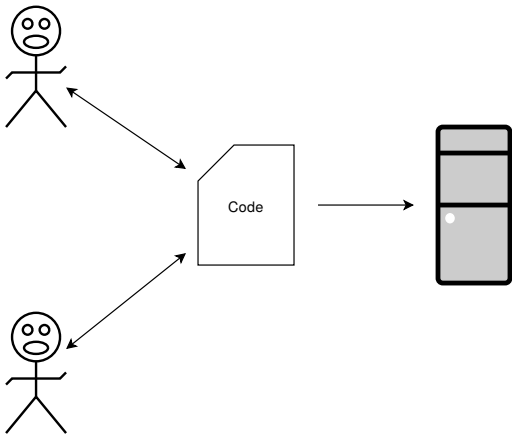


We want to control the computer. Code is a tool to achieve this.

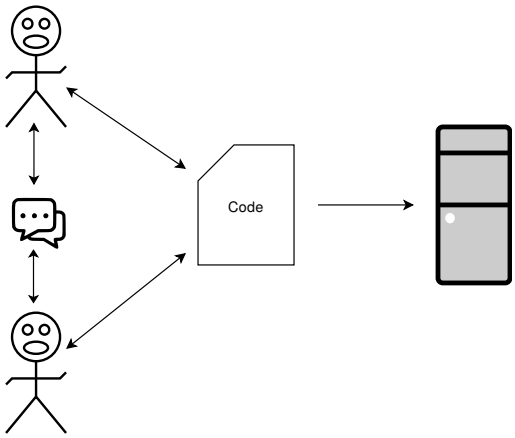
What is Code?



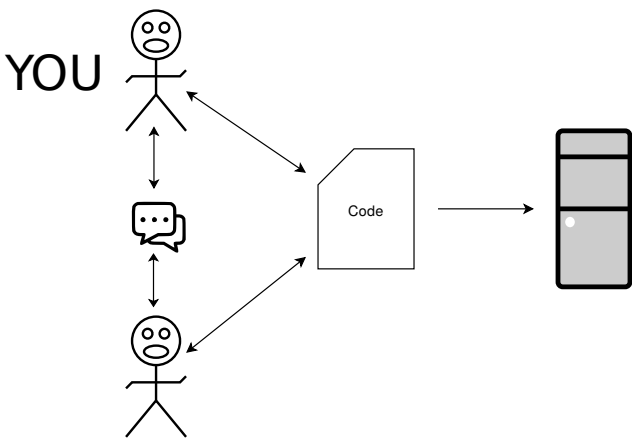
What is Code?



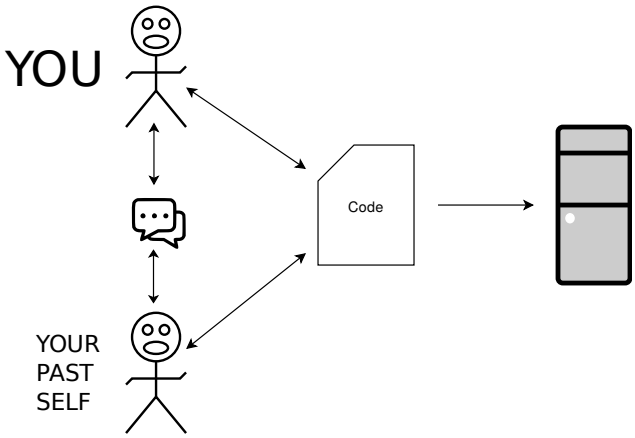
What is Code?



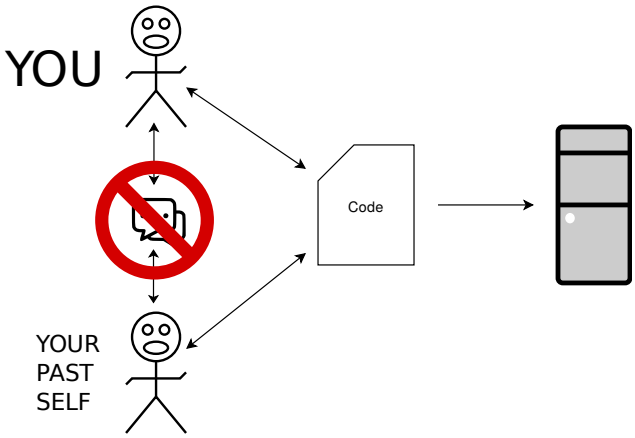
What is Code?



What is Code?



Why to write Beautiful Code?



Why to write Beautiful Code?

> "πάντα χωρεῖ καὶ οὐδὲν μένει" καὶ "δὶς ἐς τὸν αὐτὸν ποταμὸν οὐκ ἂν ἐμβαίης"

Heraclitus of Ephesus, ~500 BC

This roughly translates to:

> You cannot talk to your past self about code he or she wrote.

How to write Betautiful Code? Naming!

Do you know what this is doing?

In [6]:
```
twitter_search('obama', numtweets=0, retweets=False, unicode=True)
```

```
Way better! Great! 👍
```

In [7]:
```
twitter_search('obama', numtweets=20, retweets=False, unicode=True)
```

```
Way better! Great! 👍
```

Source: [Beyond PEP 8 - Best practices for beautiful intelligible code - PyCon 2015](https://www.youtube.com/watch?v=wf-BqAjZb8M) by Raymond Hettinger / [@raymondh](https://twitter.com/raymondh)

How to write Betautiful Code? Naming!

Part 2.

In [9]:
```
with open('data-samples/turbine_models.csv', 'r') as f:
    turbine_models = f.read().splitlines()
```

In [10]:
```
model_names = np.unique(turbine_models)
```

In [11]:
```
model_names
```

Out[11]:
```
array(['1.5-70.5', '1.5-77', '1.5SLE', '1.6-100', '1.7-100', '1.7-77',
       '1.85-87', '108', '110', '120', '15/50', '15S', '180', '2.0-116',
       '2.3-116', '225/30', '23E', '23_160', '250', '250/50', '250KW',
       '29-STALL-225', '33M_VS', '40/500', '47-750', '500/41', '54-750',
       '56-100', '56-900', '60', '65', '65/13', '65kW', '75/L7', '95T',
       'A-1500', 'AOC15/50', 'AOC15_65', 'AW116/3000', 'AW125/3000',
       'AW125/3150', 'AW77/1500', 'AW82/1500', 'AWE54-900', 'B62', 'C89',
       'C93', 'C96', 'C99', 'CCWE3.60-116', 'D8.2', 'D9.2', 'DW-52-750',
       'DW-54-900', 'DW52_900', 'E-3120', 'E3120', 'ECO 86', 'EW 1.5S',
       'EW50', 'FD-77-1500', 'FL 1000', 'FL100', 'FL1500', 'FL250',
       'FL2500', 'G114-2.0', 'G114-2.1', 'G52-0.8', 'G58-0.85', 'G80',
       'G80-2.0', 'G83-2.0', 'G87', 'G87-2.0', 'G90-2.0', 'G97',
       'G97-2.0', 'G97-2.1', 'GE1.5-65', 'GE1.5-70.5', 'GE1.5-77',
       'GE1.5-82.5', 'GE1.5-87', 'GE1.5-91', 'GE1.5sle', 'GE1.6-100',
       'GE1.6-77', 'GE1.6-82.5', 'GE1.6-87', 'GE1.62-100', 'GE1.62-103',
       'GE1.62-82.5', 'GE1.62-87', 'GE1.62-91', 'GE1.68-82.5',
       'GE1.7-100', 'GE1.7-103', 'GE1.715-103', 'GE1.79-100',
       'GE1.85-82.5', 'GE1.85-87', 'GE2.0-116', 'GE2.1-116', 'GE2.3-107',
       'GE2.3-116', 'GE2.4-107', 'GE2.5-100', 'GE2.5-116', 'GE2.5-120',
       'GE2.85-103', 'GEV MP', 'GEV MP-R', 'GW100', 'GW109', 'GW136',
       'GW70', 'GW82', 'GW87', 'H111-2000', 'H93-2000', 'HQ1650',
       'HQ2000', 'Haliade 150-6', 'K100', 'KVS33', 'LTW-77', 'M530',
       'M700_225', 'MHI600', 'MK1', 'MM92', 'MS2', 'MWT-600-45',
       'MWT1000', 'MWT1000A', 'MWT102/2.4', 'MWT57/1.0', 'MWT600',
       'MWT62/1.0', 'MWT92/2.4', 'MWT95/2.4', 'Micon 100',
       'Multi-power 44', 'N100', 'N1000', 'N1000_59', 'N117', 'N43',
       'N54', 'N54/1000', 'N60', 'N90', 'NM-19', 'NM44', 'NM48',
       'NM48_600', 'NM48_750', 'NM52', 'NM52_900', 'NM54', 'NM54_950',
       'NM72', 'NM72C', 'NM72_1500', 'NM82', 'NPS 100', 'NPS 100-21',
       'NPS-100', 'NPS-2300', 'NTK 150', 'NTK 65', 'NTK 65/13', 'NTK65',
       'NW100', 'NW100_19', 'PS-600', 'PS-600b', 'PW56', 'S2.5-90', 'S64',
       'S64-1.25', 'S88', 'S95', 'S97', 'SE10020E', 'SE10520E',
       'SE8720IIIE', 'SE9320IIIE', 'SI.250', 'SL1500', 'SWT-2.3-101',
       'SWT-2.3-108', 'SWT-2.3-82', 'SWT-2.3-93', 'SWT-2.346-108',
       'SWT-2.35-108', 'SWT-2.37-108', 'SWT-2.625-120', 'SWT-3.0-101',
       'SWT-3.2-113', 'SWT1.3_62', 'SWT2.3_101', 'SWT2.3_93', 'Sojets',
       'T-400-34', 'T-600-48', 'T600-48DS', 'U57', 'UP1500', 'V100-1.8',
       'V100-1.82', 'V100-2.0', 'V110-2.0', 'V112-3.0', 'V112-3.075',
       'V112-3.3', 'V117-3.3', 'V117-3.45', 'V126-3.0', 'V126-3.3',
       'V126-3.45', 'V15', 'V17', 'V27', 'V34', 'V39', 'V39-0.5', 'V42',
       'V42-0.65', 'V44', 'V44-0.6', 'V47', 'V47-0.6', 'V47-0.66',
       'V47-0.71', 'V66', 'V66-1.65', 'V7', 'V80-1.8', 'V80-2.0', 'V82',
       'V82-1.65', 'V90-1.8', 'V90-1.86', 'V90-3.0', 'Vensys64',
       'Vensys77', 'Vensys82', 'W2320', 'W3000', 'WES18', 'WM15S',
       'WM17S', 'WWG0600', 'XL50', 'Z40', 'Z48', 'Z50', 'missing'],
      dtype='<U14')
```

How to write Betautiful Code? Naming!

Part 2.

In [13]:
```
turbine_unique_result = np.unique(turbine_models, return_inverse=True, return_counts=True)
```

In [14]:
```
pd.DataFrame({
    'turbine_models': turbine_models,
    'counts': turbine_unique_result[2][turbine_unique_result[1]]
}).sample(10)
```

Out[14]:

	turbine_models	counts
39225	N100	191
45670	GE2.5-100	338
9654	MWT62/1.0	1820
11484	GE1.7-100	1318
30366	NM54	35
57122	missing	4469
39869	GE1.5-77	8562
40530	GE1.5-87	465
10240	GE1.5-77	8562
24328	SWT-2.3-101	1086

How to write Betautiful Code? Naming!

Part 2, but a bit nicer.

In [16]:
```
model_names, inverse_idcs, counts = np.unique(turbine_models, return_inverse=True, return_co
unts=True)
```

In [17]:
```
pd.DataFrame({
    'turbine_models': turbine_models,
    'counts': counts[inverse_idcs]
}).sample(10)
```

Out[17]:

	turbine_models	counts
39225	N100	191
45670	GE2.5-100	338
9654	MWT62/1.0	1820
11484	GE1.7-100	1318
30366	NM54	35
57122	missing	4469
39869	GE1.5-77	8562
40530	GE1.5-87	465
10240	GE1.5-77	8562
24328	SWT-2.3-101	1086

How to write Betautiful Code? Named containers!

What if one wants to keep the elements of a tuple bound together?

Containers!

- dictionary: no schema, can be dynamically changed (this is mostly a disadvantage)
- namedtuple (https://docs.python.org/3/library/collections.html#collections.namedtuple): like dict, but syntax `foo.parameter` instead of `foo['parameter']` and fixed schema
- dataclass (https://docs.python.org/3/library/dataclasses.html) for Python >= 3.7
- attrs (https://www.attrs.org/en/stable/): like dataclasses (not part of core Python, but also for < 3.7)
- write your own class

How to write Betautiful Code? Naming!

Don't try this at home!

In [19]:
```
(ツ) = ಠ_ಠ
```

In [20]:
```
(ツ)
```

Out[20]:
```
¯\_(ツ)_/¯
```

In [21]:
```
(ツ) = ʏᵒᵘᴏᴺˡʸʟⁱᵛᵉᴏᴺᶜᵉ
```

In [22]:
```
(ツ)
```

Out[22]:
```
( ͡° ͜ʖ ͡°)
```

This is valid Python code! Can anybody imagine how this works?

inspired by https://twitter.com/yennycheung/status/1099349853518397440 @ [#pythonpizza] (https://berlin.python.pizza/)

How to write Betautiful Code? Naming!

If you need to name a *thing*, can you describe the thing to somebody who has no idea what it is and what it does with a single precise word?

- neat convention: encode the unit in the name, e.g. `distance_km`
- docstrings and comments: don't repeat the code, focus on the non-obvious
- if you can't give it a good name, it might be an indication of bad abstraction

How to write Betautiful Code? Naming!

1. avoid abbreviations
2. don't be too generic
3. don't be too specific
4. names should not be too long
5. names should not be meaningless

Ad (1) and (4): think twice if these before using these names:

`data, value, controller, manager, tmp, helper, util, tool, x, a, foo`

How to write Betautiful Code? Naming!

- use functions to name parts of your code
  - only 10-30 lines of code in each function
  - also makes scope smaller with a clear interface
  - also reduces indentation
- avoid so called magic values, put numbers in constants:

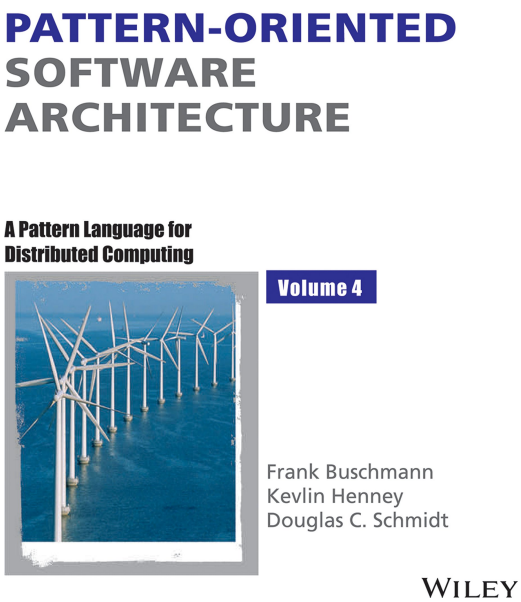
In [24]:
```
# this is made up out of thin air, but at least documented
MY_ARBITRARY_THRESHOLD = 23.2
```

How to write Betautiful Code? Naming!

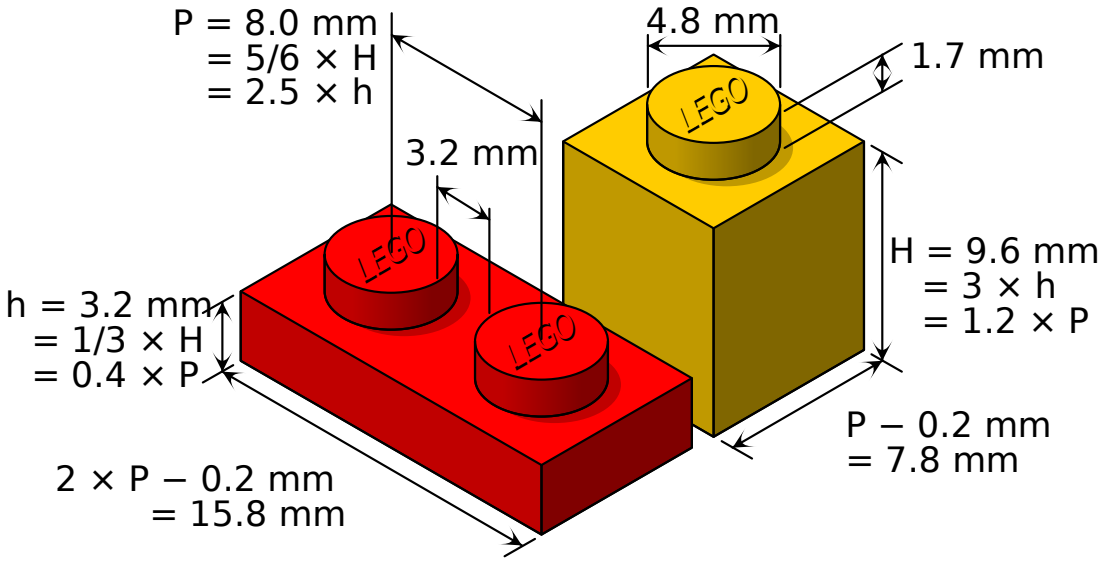
Always remember:

- Be a poet!
- Naming is difficult!

How to write Beautiful Code? Pattern!



Break the problem into smaller blocks



Source: https://de.wikipedia.org/wiki/Lego#/media/File:Lego_dimensions.svg CC BY-SA 3.0

Stateless Blocks

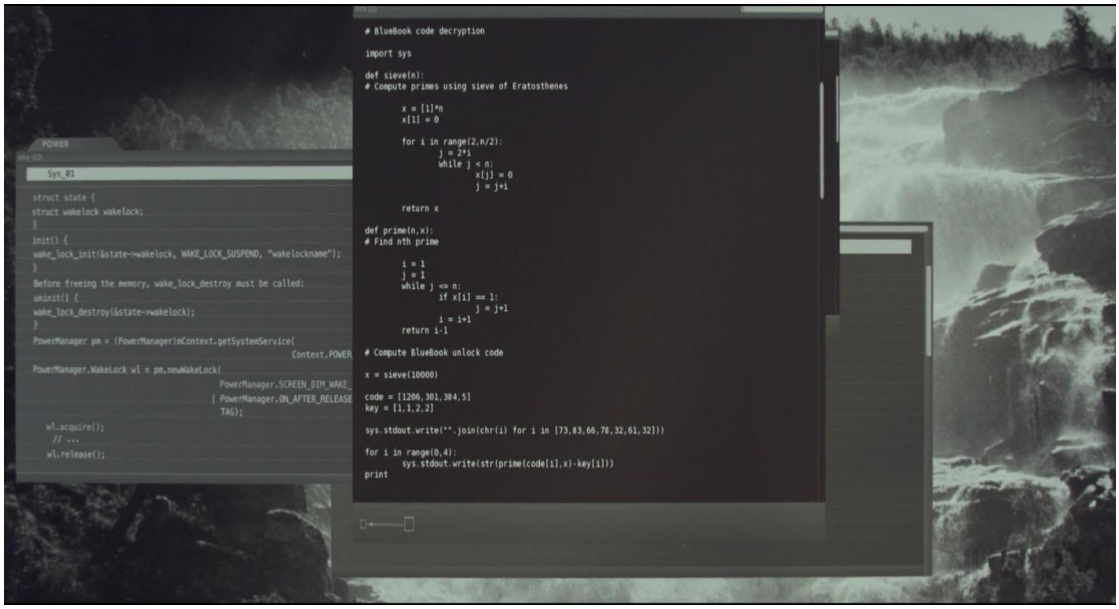
They can be complex inside and but should be simple to use from outside.

Break the problem into smaller blocks, use them to construct larger blocks. Each block can be complicated inside, but need to have a clear interface which allows to connect them. If we know what the block does, we don't need to care about it's implementation as long as each input gives an deterministic output, i.e. the output depends only on the input and not on some other state ("stateless"). Smaller blocks represent the low level abstraction, larger blocks of smaller blocks are the high level abstraction. Low level is closer to the hardware and allows a more fine grained control. High level is easier ind simpler and closer to the problem definition.

If we know input and output and what the block does, it can be used without caring about the inside. We can use the abstraction to forget about things and concentrate on other parts. It can be tested and debugged easier. If the output does not depend only on its input, but one some other state (not stateless!), things get more complicated. How to test and debug it? We need to look at more code at once. Pieces of code are more entangled. That's also what makes working with data more complicated than normal software programming (you have to manage the data (state!), not only the routines which handle data).

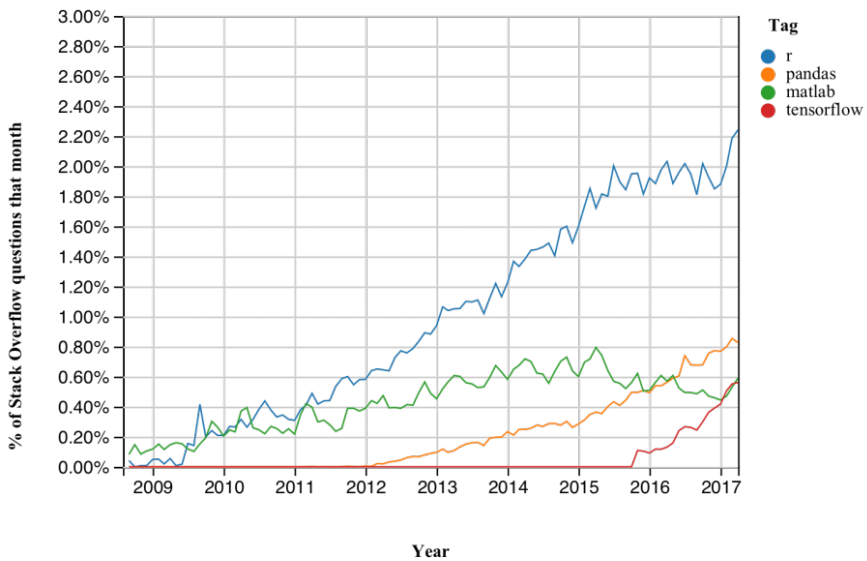
A good abstraction has different layers, higher layers use only lower layers. Therefore cyclic dependencies are avoided automatically.

Why Python?



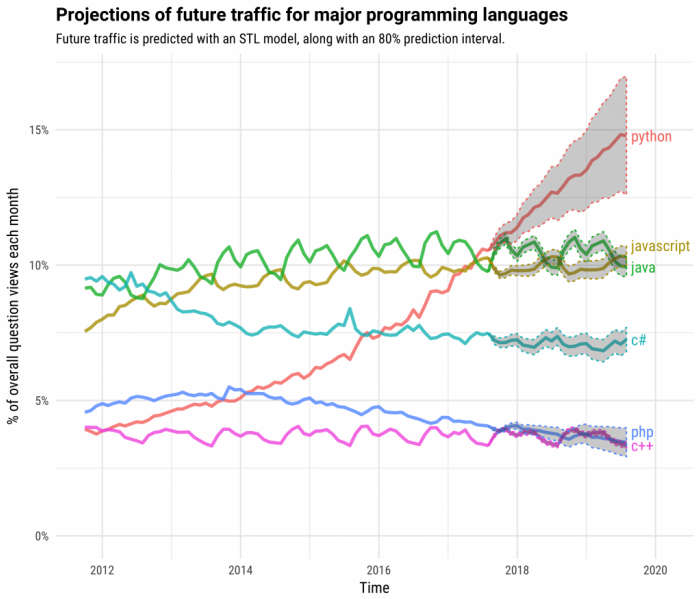
Source: [Screenshot](http://i.imgur.com/C44iJeR.jpg) of the movie [Ex Machina](https://www.imdb.com/title/tt0470752/), see also https://www.reddit.com/r/movies/comments/365f9b/secret_code_in_ex_machina/

Why Python?



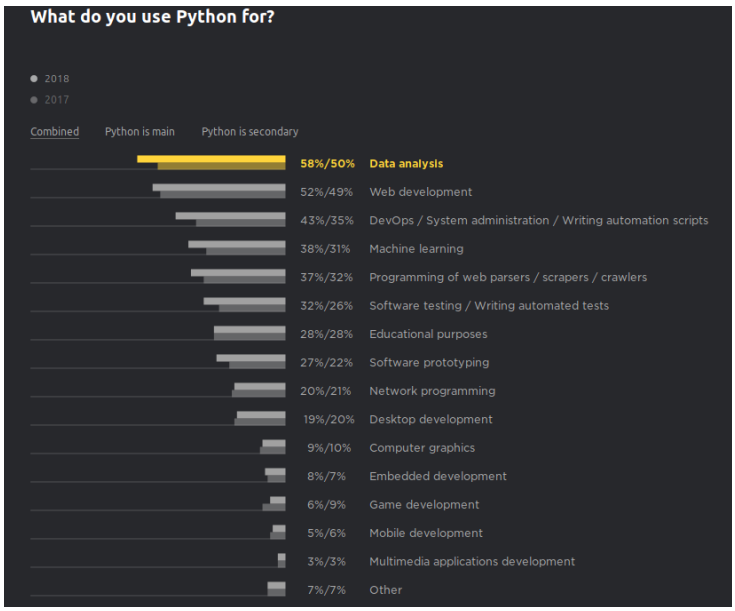
Source: https://stackoverflow.blog/2017/05/09/introducing-stack-overflow-trends/

Why Python?



Source: https://stackoverflow.blog/2017/09/06/incredible-growth-python/

Why Python?



Source: https://www.jetbrains.com/research/python-developers-survey-2018/

Why Python?

Python is free software!

- free* (as in freedom, not as in free beer)
- you can look inside and modify/fix things yourself
- huge and welcoming community

* GPL compatible, but not GPL not sure if RMS would call this free

PEP 8

= Official Style Guide for Python Code

- where to put spaces
- strings and docstrings
- naming and snake_case vs CamelCase

Typical disagreements:

- line width ("79 is too short")
- quoting styles (' vs ")
- tabs vs spaces

PEP 8

Extensions of PEP 8 for docstrings and parameters:

- rst (https://sphinxcontrib-napoleon.readthedocs.io/en/latest/index.html)
- numpy (https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_numpy.html)
- Google (https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html)
- Epytext (http://epydoc.sourceforge.net/manual-epytext.html)

PEP 8

PEP 8 example code
Source: https://gist.github.com/RichardBronosky/454964087739a449da04

In [ ]:

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-
"""This module's docstring summary line.

This is a multi-line docstring. Paragraphs are separated with blank lines.
Lines conform to 79-column limit.

Module and packages names should be short, lower_case_with_underscores.
Notice that this in not PEP8-cheatsheet.py

Seriously, use flake8. Atom.io with https://atom.io/packages/linter-flake8
is awesome!

See http://www.python.org/dev/peps/pep-0008/ for more PEP-8 details
"""


import os  # STD lib imports first
import sys  # alphabetical

import some_third_party_lib  # 3rd party stuff next
import some_third_party_other_lib  # alphabetical

import local_stuff  # local stuff last
import more_local_stuff
import dont_import_two, modules_in_one_line  # IMPORTANT!
from pyflakes_cannot_handle import *  # and there are other reasons it should be avoided # n
oqa
# Using # noqa in the line above avoids flake8 warnings about line length!


_a_global_var = 2  # so it won't get imported by 'from foo import *'
_b_global_var = 3

A_CONSTANT = 'ugh.'


# 2 empty lines between top-level funcs + classes
def naming_convention():
    """Write docstrings for ALL public classes, funcs and methods.

    Functions use snake_case.
    """
    if x == 4:  # x is blue <== USEFUL 1-liner comment (2 spaces before #)
        x, y = y, x  # inverse x and y <== USELESS COMMENT (1 space after #)
    c = (a + b) * (a - b)  # operator spacing should improve readability.
    dict['key'] = dict[0] = {'x': 2, 'cat': 'not a dog'}


class NamingConvention(object):
    """First line of a docstring is short and next to the quotes.

    Class and exception names are CapWords.

    Closing quotes are on their own line
    """

    a = 2
    b = 4
    _internal_variable = 3
    class_ = 'foo'  # trailing underscore to avoid conflict with builtin
```

PEP 8

Use a code linter in your editor and tests!

- pycodestyle (https://pypi.org/project/pycodestyle/) (formerly called "pep8"): checks only style, not validity
- flake8 (https://github.com/PyCQA/flake8): faster than pylint, a combination of (pycodestyle and pyflakes)
- pylint (https://www.pylint.org/): stricter than flake8
- black (https://github.com/ambv/black): code formatter

On command line:

```
$ pylint3 debug-numpy-linalg-norm.py
************* Module debug-numpy-linalg-norm
debug-numpy-linalg-norm.py:1:0: C0103: Module name "debug-numpy-linalg-norm" doesn't confor
m to snake_case naming style (invalid-name)
debug-numpy-linalg-norm.py:1:0: C0111: Missing module docstring (missing-docstring)
debug-numpy-linalg-norm.py:3:0: C0103: Constant name "d" doesn't conform to UPPER_CASE nami
ng style (invalid-name)

-----------------------------------
Your code has been rated at 0.00/10
```

Questions?

PEP 8: Exercise

Try out code one or more code linters and fix some PEP 8 issues!

Suggestion:

- `exmachina.py` (many violations)
- a well known module (e.g. numpy, logging, ...)
- your own code

Everything is an Object

In [26]:
```
meaning = 42
```

In [27]:
```
from datetime import datetime
```

In [28]:
```
birthdays = {
    'Alice': datetime(1978, 2, 1),
    'Bob': datetime(1978, 2, 3)
}
```

In [29]:
```
data = [1, 2, 3, meaning, birthdays]
```

In [30]:
```
del birthdays    # del is very rarely needed in real life, this is for demonstration
```

In [31]:
```
data
```

Out[31]:
```
[1,
 2,
 3,
 42,
 {'Alice': datetime.datetime(1978, 2, 1, 0, 0),
  'Bob': datetime.datetime(1978, 2, 3, 0, 0)}]
```

Everything is an Object

A function takes objects as inputs and its return value is an object. Functions are objects too, everything is an object!

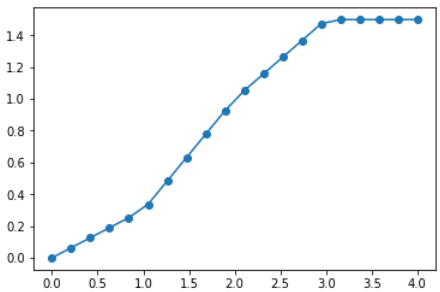
In [33]:
```
power_curve = interp1d(
    [0, 1, 2, 3, 4],
    [0, 0.3, 1., 1.5, 1.5]
)
```

In [34]:
```
power_curve(1.5)
```

Out[34]:
```
array(0.65)
```

In [35]:
```
def plot_func(func):
    x = np.linspace(0, 4, num=20)
    y = func(x)
    plot(x, y, 'o-')

plot_func(power_curve)
```



Everything is an Object

See also: https://docs.python.org/3/reference/datamodel.html

- in Python data is stored in objects - and everything is an object (also functions, modules, ...)
- objects can be stored in variables (aka "names") or in other objects (nested)
- variables/names are created via assignment = or other Python statements (`import`, `def`, `class`, ...)

Everything is an Object

Every object consists of:

- **identity**: never changes after creation (like a pointer or memory address)
- **value**: the data to be stored, something like list elements (can be changed)
- **type**: e.g. list, int, float, dict, ... (better not try to change this!)

Everything is an Object

- variables contain only references to the object (the identity)
- assignments and parameters to functions don't copy objects, only pass references
- there are types of objects which contain references to other objects (`list`, `dict`, `tuple`, ...)
- some types cannot contain other objects (`str`, `int`, `float`, ...)
- some operations modify objects, other operations create new objects

Everything is an Object: Assignment and Modification

In [36]:
```
list1 = [1,2,3]
list2 = [1,2,3]
another_list1 = list1
```

In [37]:
```
list1 == list2
```

Out[37]: True

In [38]:
```
list1 is list2
```

Out[38]: False

In [39]:
```
list1 is another_list1
```

Out[39]: True

Everything is an Object: Assignment and Modification

In [40]:
```
list1.append(42)  # modifies list1
list1
```

Out[40]: [1, 2, 3, 42]

In [41]:
```
list2
```

Out[41]: [1, 2, 3]

In [42]:
```
another_list1
```

Out[42]: [1, 2, 3, 42]

Everything is an Object: Assignment and Modification

In [43]:
```
merged_lists = list1 + list2  # creates a new object!
merged_lists
```

Out[43]:
```
[1, 2, 3, 42, 1, 2, 3]
```

`list1` and `another_list1` are identical, i.e. their identity is the same:

In [44]:
```
print("id(list1) =", id(list1))
print("id(list2) =", id(list2))
print("id(another_list1) =", id(another_list1))
print("id(merged_lists) =", id(merged_lists))
```

```
id(list1) = 139889121097608
id(list2) = 139889086691272
id(another_list1) = 139889121097608
id(merged_lists) = 139889123752008
```

Everything is an Object: Assignment and Modification

In [45]:
```
list1 is another_list1
```

Out[45]:
```
True
```

In [46]:
```
list1 == another_list1
```

Out[46]:
```
True
```

In [47]:
```
list1 = [1, 2, 3]
```

In [48]:
```
another_list1
```

Out[48]:
```
[1, 2, 3, 42]
```

In [49]:
```
list1 is another_list1
```

Out[49]:
```
False
```

Everything is an Object: Assignment and Modification

Summery: don't confuse the following:

- creation of a new object: something like `[1,2,3]`, 23 or `np.array([1,2,3])`
- modifying of an existing object: `list1.append(42)`
- assignment: assigns a reference to the variable on lhs of the =

If a method (or function) like `list.append()` modifies the object or returns a new one is different for each method.

Everything is an Object: Copy an object

Sometimes an object needs to be copied:

In [51]:
```
import copy
```

In [52]:
```
first_list = [1, 2, 3]
copy_of_first_list = copy.copy(first_list)
```

Nested objects are copied with `copy.deepcopy()`.

Everything is an Object: Classes

New types are created by implementing a class:

In [53]:
```
class Polynomial(tuple):
    """Something like 3*x² + x."""
    ...
```

Everything is an Object: Classes

Let's create some objects of our new type:

In [54]:
```
quadratic_polynomial = Polynomial((3, 2, 0))
linear_polynomial = Polynomial((0, 2, 1))
```

In [55]:
```
quadratic_polynomial
```

Out[55]:
```
(3, 2, 0)
```

In [56]:
```
def add_polynomials(polynomial1, polynomial2):
    # FIXME this is broken for polynomials of different degree
    # zip will take consider only shorter iterable
    return Polynomial((coeff1 + coeff2
                       for coeff1, coeff2 in zip(polynomial1, polynomial2)))
```

In [57]:
```
add_polynomials(quadratic_polynomial, linear_polynomial)
```

Out[57]:
```
(3, 4, 1)
```

Everything is an Object: Classes

Classes can be considered as name space:

In [58]:
```
class Polynomial(tuple):
    """Something like 3*x² + x."""

    def add(polynomial1, polynomial2):
        # FIXME this is broken for polynomials of different degree
        # zip will take consider only shorter iterable
        return Polynomial((coefficient1 +  coefficient2
                           for coefficient1, coefficient2 in zip(polynomial1, polynomial2)))
```

In [60]:
```
Polynomial.add(quadratic_polynomial, linear_polynomial)
```

Out[60]: (3, 4, 1)

Python knows that `quadratic_polynomial` is of type `Polynomial`, so a shorter (mostly) equivalent way of the same line is:

In [61]:
```
quadratic_polynomial.add(linear_polynomial)
```

Out[61]: (3, 4, 1)

Everything is an Object: Classes

By convention the first parameter in class methods is called `self`, you should stick to this convention. It's role is similar to `this` in C++ or Java.

In [62]:
```
class Polynomial(tuple):
    """Something like 3*x² + x."""

    def add(self, other):
        # FIXME this is broken for polynomials of different degree
        # zip will take consider only shorter iterable
        return Polynomial((self_coeff + other_coeff
                           for self_coeff, other_coeff in zip(self, other)))
```

Everything is an Object: Classes

In [63]:
```
class Polynomial(tuple):
    """Something like 3*x² + x."""

    def __add__(self, other):
        # FIXME this is broken for polynomials of different degree
        # zip will take consider only shorter iterable
        return Polynomial((self_coeff + other_coeff
                           for self_coeff, other_coeff in zip(self, other)))
```

There is a very consistent protocol to modify how things behave in Python using so called "dunder" methods starting and ending with two underscores `__do_something__`.

More here: https://www.youtube.com/watch?v=cKPlPJyQrt4 (https://www.youtube.com/watch?v=cKPlPJyQrt4)

Everything is an Object: Classes

A different view on classes is a bit more common: classes are like Platotic forms. They define how objects are created (constructor in __new__ and __init__), how they store data (in their attributes) and how they behave (i.e. which methods do they implement).

In [66]:
```
class QuadraticPolynomial(Polynomial):    # inherits from a Polynomial = is a special case o
f a Polynomial
    def __init__(self, coefficients):     # __init__ is called to initialize new objects aft
er creation
        self.degree = len(coefficients)

    def __repr__(self):
        return f"{self[0]}x² + {self[1]}x + {self[2]}"

    def __call__(self, x):
        return sum(coeff * x**i for i, coeff in enumerate(reversed(self)))

    def __add__(self, other):
        # FIXME this is broken for polynomials of different degree
        # zip will take consider only shorter iterable
        return Polynomial((self_coeff + other_coeff
                           for self_coeff, other_coeff in zip(self, other)))
```

In [67]:
```
quadratic_polynomial = QuadraticPolynomial(quadratic_polynomial)
quadratic_polynomial
```

Out[67]:
```
3x² + 2x + 0
```

Attributes not only used for methods, but also used to store non-callable objects:

In [68]:
```
quadratic_polynomial.degree
```

Out[68]:
```
3
```

Everything is an Object

To summarize: objects are created either...

...by calling classes (similar to functions):

In [69]:
```
int("42")
```

Out[69]:
```
42
```

In [70]:
```
dict(key1=42, key2=43)
```

Out[70]:
```
{'key1': 42, 'key2': 43}
```

Everything is an Object

To summarize: objects are created either...

...by literals:

In [71]:
```
42
```

Out[71]: 42

In [72]:
```
{'key': 42, 'key': 43}
```

Out[72]: {'key': 43}

Everything is an Object

To summarize: objects are created either...

...by statements, but this is a special case and only important to emphasize that everything is an object:

In [73]:
```
import logging

def my_function():
    pass

class SomeClass:
    pass
```

Everything can be modified

In [75]:
```
def evil_print(*value, sep=' ', end='\n', file=None, flush=False):
    return "(͡≖ ͜ʖ͡≖)"
```

In [76]:
```
print("hello world")
```

```
hello world
```

In [77]:
```
print = evil_print
```

In [78]:
```
print("hello world")
```

Out[78]: '(͡≖ ͜ʖ͡≖)'

Everything can be modified: But why?

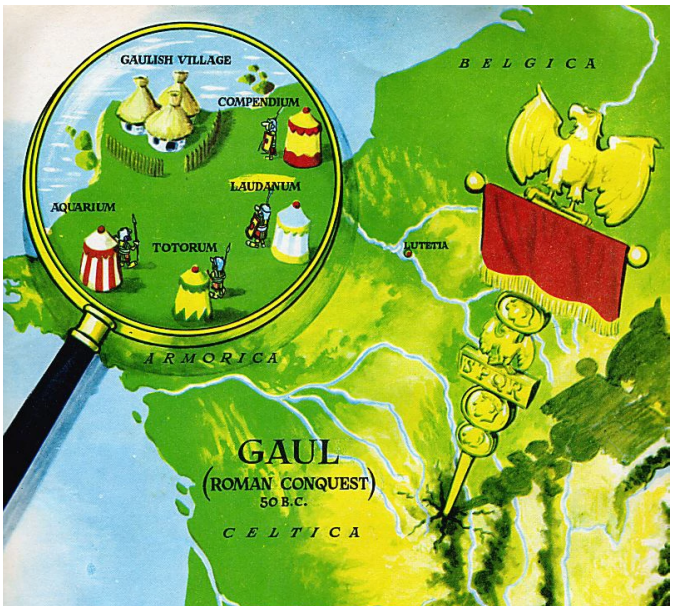
- monkey patching can help if you *really* need to modify 3rd code (e.g. bug-fix)
- temporary experiments if you cannot restart the Python process or so
- be careful with built-ins (and keywords), syntax high-lighting helps

In [81]:
```
list = list((1,2,3))
```

In [82]:
```
list
```

Out[82]: [1, 2, 3]

Really everything can be modified?



Immutable types and the traps of mutability

immutable types:

`str, int, float, tuple, frozenset, NoneType`

almost everything else is mutable, especially:

`list, dict`

See also: https://docs.python.org/3/reference/datamodel.html

Immutable types and the traps of mutability

In [84]:
```
def extend_list(element, l=[]):
    l.append(element)
    return l
```

In [85]:
```
extend_list(4, [1,2,3])
```

Out[85]: [1, 2, 3, 4]

In [86]:
```
extend_list(1)
```

Out[86]: [1]

In [87]:
```
extend_list(1)
```

Out[87]: [1, 1]

Immutable types and the traps of mutability

Never use mutable objects as default arguments and avoid modifying input parameters (unless you need to avoid copying a kit of data).

In [88]:
```
def extend_list(element, l=None):
    if l is None:
        l = []  # note that l is not modyfied, but a new object is assigned
    l.append(element)
    return l
```

Scope: Packages, Modules, Classes and Functions

Quiz: valid Python code?

In [89]:
```
for i in range(3):
    def meaning(n):
        return 42
```

In [90]:
```
class Life:
    for i in range(3):
        def meaning(n):
            return 42
```

In [91]:
```
for i in range(3):
    class Life:
        for i in range(3):
            def meaning(n):
                return 42
```

https://docs.python.org/3/reference/executionmodel.html (https://docs.python.org/3/reference/executionmodel.html)

Scope: Packages, Modules, Classes and Functions

Quiz: order of execution - what will happen here?

In [92]:
```
print_meaning()

def print_meaning():
    print(42)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-92-33ceca181ff9> in <module>
----> 1 print_meaning()
      2
      3 def print_meaning():
      4     print(42)

NameError: name 'print_meaning' is not defined
```

In [93]:
```
def call_print_meaning():
    print_meaning()

def print_meaning():
    print(42)

print_meaning()
```

```
42
```

Scope: Packages, Modules, Classes and Functions

Quiz: order of execution - what will happen here?

In [94]:
```
def some_function():
    print(not_defined_variable + 2)
```

Scope: Packages, Modules, Classes and Functions

In [95]:
```
MY_CONSTANT = 42

def some_function():
    fancy_calculation = 1 * MY_CONSTANT
    return fancy_calculation
```

In [96]:
```
some_function()
```

Out[96]:
```
42
```

Scope: Packages, Modules, Classes and Functions

In [97]:
```
MY_CONSTANT = 42

def some_function():
    fancy_calculation = 1 * MY_CONSTANT

    def inner_function():
        return 0.5 * fancy_calculation

    fancy_calculation = inner_function()
    return fancy_calculation
```

In [98]:
```
some_function()
```

Out[98]:
```
21.0
```

Scope: Packages, Modules, Classes and Functions

In [99]:
```
some_list = []

def extend_list(n):
    some_list.append(n)
    return some_list

extend_list(1)
```

Out[99]: [1]

In [100]:
```
some_list = []

def extend_list(n):
    some_list.append(n)
    if len(some_list) > 3:
        # too long...
        some_list = []
    return some_list

extend_list(1)
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-100-18e1abc6f3db> in <module>
      8     return some_list
      9
---> 10 extend_list(1)

<ipython-input-100-18e1abc6f3db> in extend_list(n)
      2
      3 def extend_list(n):
----> 4     some_list.append(n)
      5     if len(some_list) > 3:
      6         # too long...

UnboundLocalError: local variable 'some_list' referenced before assignment
```

Scope: Packages, Modules, Classes and Functions

In [ ]:
```
a = 3

class A:
    a = a + 2
    a = a + 1
    b = a
```

In [ ]:
```
a, A.a, A.b
```

Scope: Packages, Modules, Classes and Functions

In [ ]:
```
class B:
    a = 42
    b = tuple(a + i for i in range(10))
```

In [ ]:
```
a, B.a
```

In [ ]:
```
B.b
```

See also: https://docs.python.org/3.3/reference/executionmodel.html (https://docs.python.org/3.3/reference/executionmodel.html)

Scope: Packages, Modules, Classes and Functions

- a name (=variable) is defined in a module, a class or a function
- names in functions are *only* visible inside this function
- names in modules are (directly) visible inside the module
- names in classes are not (directly) visible inside methods
- names in modules and classes can be accessed from outside via `module_name.variable` or `class_name.variable`

Scope: Packages, Modules, Classes and Functions

If you write a script, use a main() function to avoid to make variables local:

In [ ]:
```
MY_CONSTANTS = 42

def main():
    # fancy script code
    some_local_variable = 3
    ...

if __name__ == '__main__':
    main()
```

Imports

Looks for `logging.py` or `logging/__init__.py` in the PYTHONPATH, runs it, creates a module object and assings it to `logging`:

In [ ]:
```
import logging
```

In [ ]:
```
from logging import getLogger
```

...is mostly equivalent to:

In [ ]:
```
import logging
getLogger = logging.getLogger
```

Questions?

Exercise:

Choose:

- Write a terrible confusing script/function/whatever by modifying something you shouldn't modify! (e.g. slow down time by a factor 2, name a list `list`, let a built-in function return something surprising, ...)
- Write a function using a dunder function `__<some name>__` doing something very useful or something very evil!
- List use cases: when does it make sense to modify something? Which objects should never be modified?

Exceptions and tracebacks

In [101]:
```
raise Exception("something bad happened")
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-101-41c2198aad14> in <module>
----> 1 raise Exception("something bad happened")

Exception: something bad happened
```

Source: https://docs.python.org/3/library/exceptions.html, https://docs.python.org/3/tutorial/errors.html

Exceptions and tracebacks

Exception chaining is powerful to avoid loosing the original cause:

In [ ]:
```
some_list = [1,2]
try:
    some_list[4] = 42
except Exception as e:
    raise RuntimeError("Failed to append meaning of life") from e
```

Exceptions and tracebacks

In [ ]:
```
try:
    1/0
finally:
    print("before everything explodes: 🎉")
```

Exceptions and tracebacks

In [ ]:
```
with open('turbine_models.csv') as f:
    1/0
```

In [ ]:
```
f.closed
```

Exceptions and tracebacks

- errors should never pass silently, make good error messages including parameters
- exceptions can be any object, but should better inherit from Exception
- KeyboardError does not inherit from Exception and won't be caught by `except Exception`

Debuggers

Debuggers help to inspect the inside of code. This can be useful for when searching for a bug, but also if you want to understand what a particular piece of code does.

There are many debuggers for Python (https://wiki.python.org/moin/PythonDebuggingTools):

- pdb (https://docs.python.org/3/library/pdb.html): shipped with the Python standard libary
- ipdb (https://github.com/gotcha/ipdb): tab completion, syntax highlighting, debugger for IPython
- PyCharm inlcudes a graphical debugger
- pdb++ (https://github.com/antocuni/pdb): similar to ipdb
- ...

One can inject code in a running process:

- pyrasite (http://pyrasite.com/)
- pyringe (https://github.com/google/pyringe)
- ...

  (Very helpful to debug dead locks and memory leaks!)

Let's debug something!

Invoke the debugger via break point in code:

In [ ]:
```
import pdb; pdb.set_trace()
```

In [ ]:
```
import ipdb; ipdb.set_trace()   # does not work in Jupyter notebooks
```

In [ ]:
```
breakpoint()  # for Python >= 3.7
```

Let's debug something!

Invoke debugger by calling the script from the debugger:

```
$ ipdb my_module.py  # ipdb3 (for Python 3) on some platforms
> /tmp/test.py(2)<module>()
      1 """This is my_module.py
----> 2 """
      3

ipdb>
```

⇒ opens debugger after the first line of code and after every exception ("post-mortem")

Let's debug something!

- program flow is interrupted (current thread/process)
- debugger prompt `ipdb>` acts like a Python terminal with additional commands
- inspect or continue program flow by using debugger commands

Debugger commands

- **w(here)** Print a stack trace
- **d(own)** Move the current frame one level down in the stack trace
- **u(p)** Move the current frame one level up in the stack trace
- **b(reak) [[filename:]lineno | function[, condition]]** set a new break point
- **c(ontinue)** continue execution until next break point is hit
- **n(ext)** execute line and jump to next line
- **s(tep)** step inside a function
- **l(ist) [first[, last]]** list source code of current file (from line `first` to line `last`)

Questions?

Exercise: debug something!

Investigate code you are not very familiar with using ipdb!

Suggestions:

- `np.linalg.norm`
- https://github.com/lumbric/lunchbot (https://github.com/lumbric/lunchbot)
- `logging` (e.g. `debug-logging-basicConfig.py`)
- `exmachina.py`

This time, *not* your own code! :)

Tests

- unit tests
- integration tests
- functional tests

Tests: Example

In [ ]:
```
def fibonacci_sequence(n):
    """Return a list of all Fibbonacci numbers up to the n-th Fibonacci number."""
    # FIXME don't use this function in real life
    if n == 1:
        return [0]
    elif n == 2:
        return [0, 1]
    sequence = fibonacci_sequence(n-1)
    return sequence + [sequence[-2] + sequence[-1]]
```

In [ ]:
```
fibonacci_sequence(6)
```

Tests: Example

In [ ]:
```
def test_fibonacci_sequence():
    assert fibonacci_sequence(1) == [0]
    assert isinstance(fibonacci_sequence(4), list)
    assert fibonacci_sequence(7)[-1] == 8
```

In [ ]:
```
test_fibonacci_sequence()
```

Tests: test runner

- py.test (https://pytest.org/)
- nosetests
- doctests
- many plugins: coverage, watch, ...

Tests: doctest

In [ ]:
```
"""This is a docstring.

Example
-------

>>> 1 + 1
2

"""
```

Tests: Continuous Integration

Continuous integration (CI) runs your tests automatically:

- Travis CI (https://travis-ci.com/) (note: travis-ci.com and travis-ci.org are different)
- Gitlab (https://about.gitlab.com/)
- Jenkins (https://jenkins.io/)

Tests: libraries

Libraries for writing tests:

- hypothesis (https://github.com/HypothesisWorks/hypothesis): helps to test the whole parameter space
- mock (https://docs.python.org/3/library/unittest.mock.html): change parts of your code which you can't test

Tests: corner cases

Source: https://twitter.com/brenankeller/status/1068615953989087232

Tests: corner cases

In [ ]:
```
fibonacci_sequence(0)
```

We forgot to test `fibonacci_sequence()` $n < 1$! Hypothesis helps to catch many important corner cases.

Tests: corner cases

In [ ]:
```
def fibonacci_sequence(n):
    """Return a list of all Fibbonacci numbers up to the n-th Fibonacci number."""
    # FIXME don't use this function in real life
    if n == 1:
        return [0]
    elif n == 2:
        return [0, 1]
    sequence = fibonacci_sequence(n-1)
    return sequence + [sequence[-2] + sequence[-1]]
```

Questions?

Tests: Exercise

Write a test testing something and use py.test to run it!

Include also wrong tests to see them failing! (This is also good practice in real life.)

Suggestions:

- `np.piecewise()`, `np.linalg.norm()`, ...
- git-game, `lunchbot`, `exmachina`
- your own code

Bonus: use coverage and/or watch plugins!

Additional Bonus: try to find already existing tests for the functions defined.

Logging

Excellent guide: https://docs.python-guide.org/writing/logging/ (https://docs.python-guide.org/writing/logging/)

Libraries may define loggers and emit log messages:

In [ ]:
```
import logging
logging.info('This is interesting, but not critial')

LOGGER = logging.getLogger(__name__)
LOGGER.critical("Uh this is critical %s", ಠ_ಠ)
```

The application or main script defines handlers and log levels:

In [ ]:
```
logging.basicConfig(filename='session2.log', level=logging.INFO)
```

Many more options to define handlers and levels: https://docs.python.org/3/howto/logging.html (https://docs.python.org/3/howto/logging.html)

See also `code-samples/logging_config.py`.

Tips & Tricks

Want to look into code, but don't know where the file is?

In [ ]:
```
import logging
logging.__file__
```

Import numpy and matplotlib:

In [ ]:
```
%pylab
```

In [ ]:
```
# for Jupyter notebooks
%pylab line

# Also nice for interactive plots:
#%pylab notebook
```

https://ipython.readthedocs.io/en/stable/interactive/magics.html (https://ipython.readthedocs.io/en/stable/interactive/magics.html)

Gotchas: Truthiness (1)

In [ ]:
```
int('10')
```

In [ ]:
```
float('1.3')
```

In [ ]:
```
bool('false')
```

Only the empty string is falsy, all other strings are truthy.

Gotchas: Truthiness (2)

In [ ]:
```
def uniform_random(low=None, high=None):
    if not low:
        low = 0.
    if not high:
        high = 1.

    if low >= high:
        raise ValueError(f"invalid values for low={low} and "
                         "high={high}, low < high required")

    # works only only Linux (and similar platforms), no Windows
    with open('/dev/urandom', 'rb') as f:
        # TODO one bit is a pretty low resolution, need more?
        random_byte = f.read(1)

    return ord(random_byte)/255. * abs(high - low) + low
```

In [ ]:
```
uniform_random()
```

Gotchas: Truthiness (2)

In [ ]:
```
uniform_random(-1, 0)
```

Testing on truthiness can be nice, but also dangerous if you don't know exactly all types of allowed objects!

In [ ]:
```
some_random_values = [uniform_random(-1, 0) for i in range(int(1e5))]
_ = hist(some_random_values, density=True)
```

Gotchas: Tuples (1)

In [ ]:
```
for name in ('Alice', 'Bob'):
    print('Name:', name)
```

In [ ]:
```
for name in ('Alice'):
    print('Name:', name)
```

Gotchas: Tuples (2)

In [ ]:
```
empty_list = []
```

In [ ]:
```
two_empty_lists = [[]]
```

In [ ]:
```
two_empty_lists
```

In [102]:
```
empty_tuple = ()
```

In [103]:
```
two_empty_tuples = (())
```

In [104]:
```
two_empty_tuples
```

Out[104]:
```
()
```

Gotchas: Tuples (3)

In [105]:
```
some_array = np.zeros((3,3)),
some_array
```

Out[105]:
```
(array([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]]),)
```

In [106]:
```
some_array + 1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-106-91c34a8a4b64> in <module>
----> 1 some_array + 1

TypeError: can only concatenate tuple (not "int") to tuple
```

A trailing comma can lead to confusing error messages.

Gotchas: Tuples (summary)

- empty list: `[]`
- list with one element: `[42]` or `[42,]`
- list with many elements: `[1,2,3]` or `[1,2,3,]`

- empty tuple: `()`
- list with one element: `(42,)` or in some case allowed `42,`
- list with many elements: `(1,2,3)` or `(1,2,3,)` or in some cases allowed `1,2,3` or `1,2,3,`

References