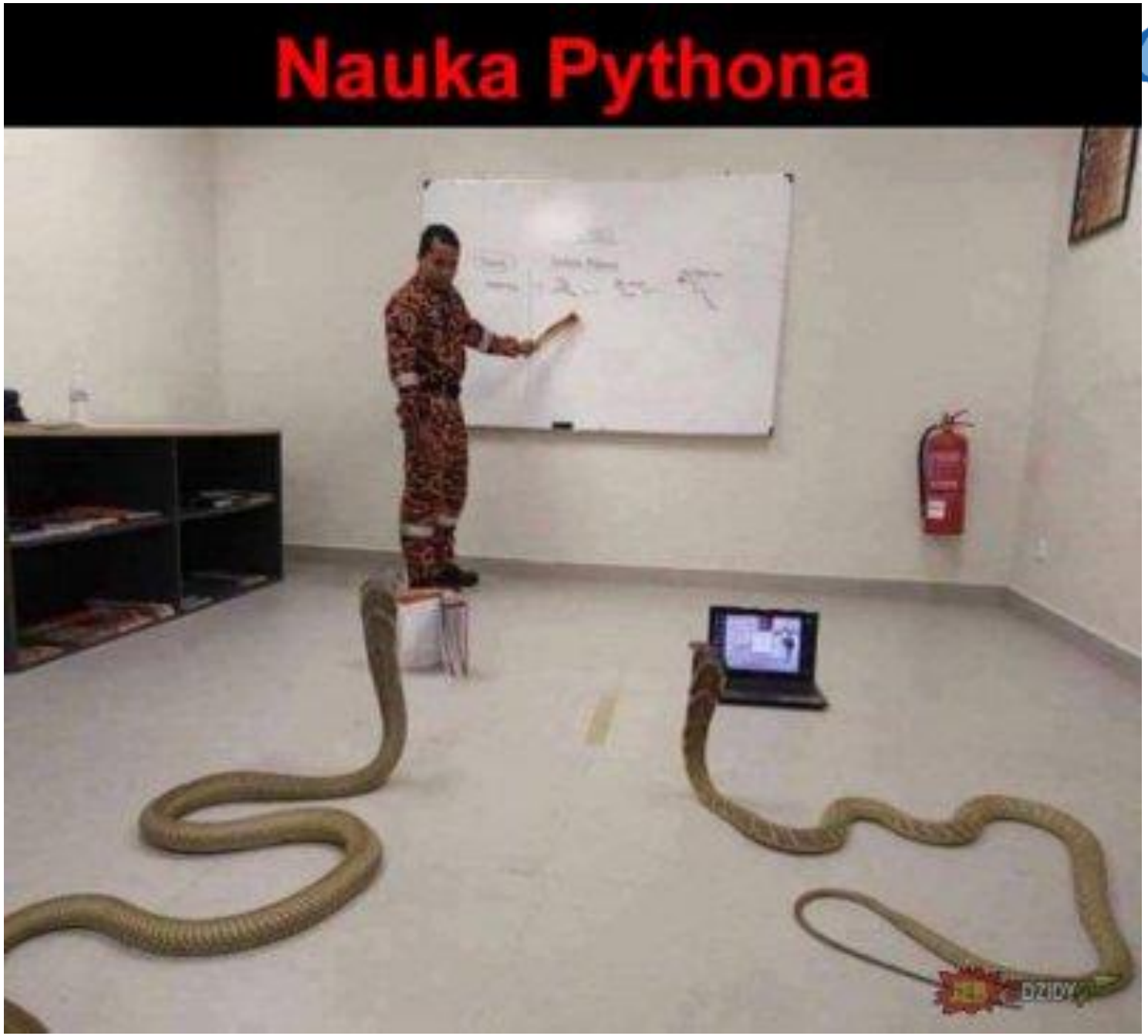
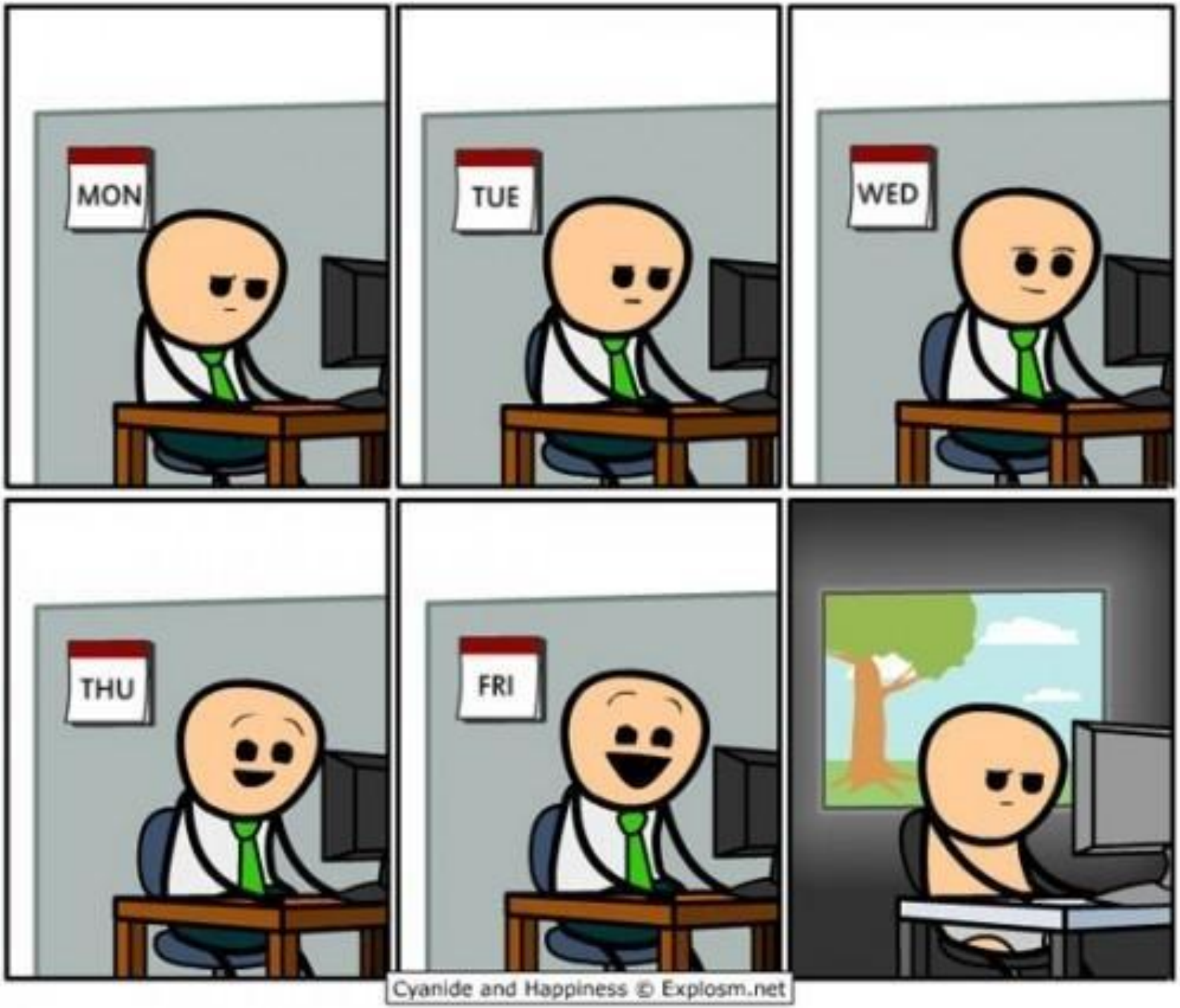
Introduction to Programming in Python

Czapiewski Paweł


Nauka Pythona


MON
TUE
WED
THU
FRI
Cyanide and Happiness © Explosm.net

AGENDA

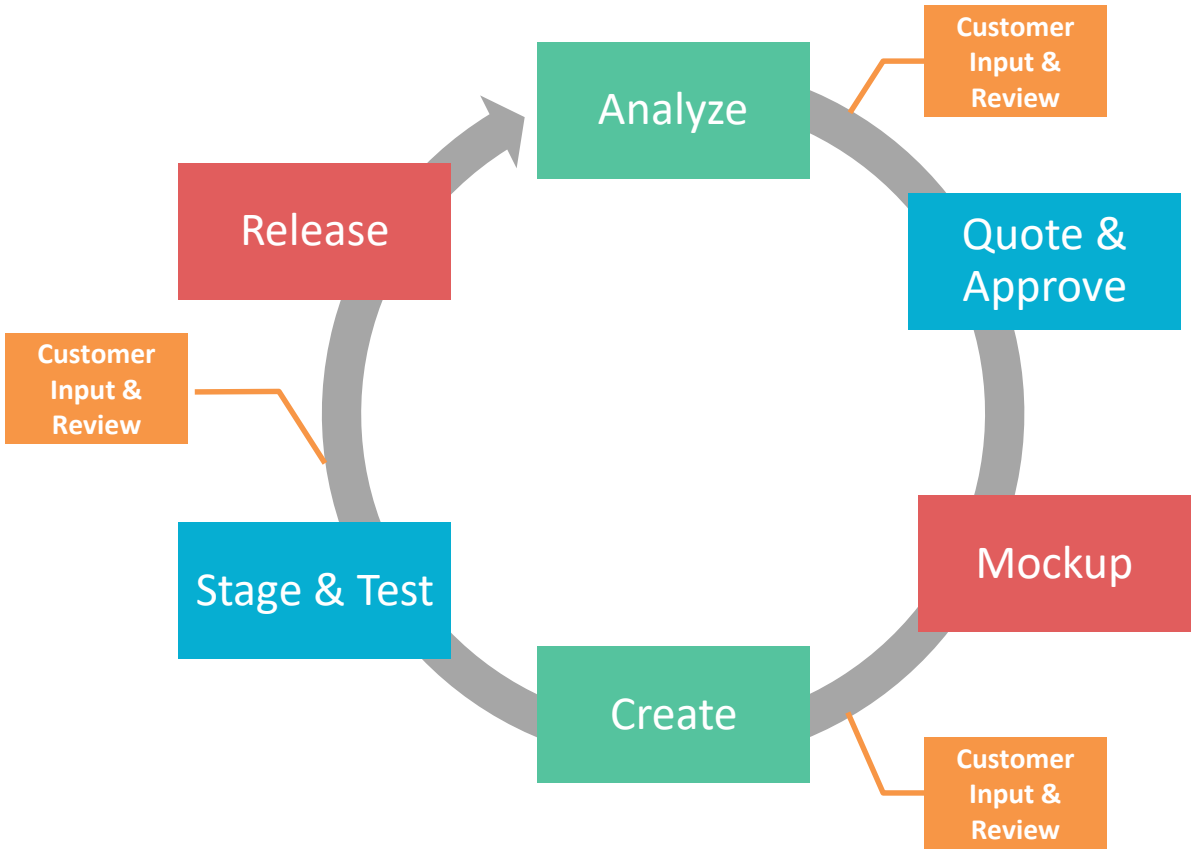

DEATH BY POWERPOINT
more awesome pictures at THEMETAPICTURE.COM

AGENDA

- Omówienie cech języka Python
- Krótki wstęp do PyCharm - podstawowe funkcje
- Wyświetlanie danych
- Uruchamianie programu z konsoli
- Tworzenia zmiennych
- Operacje na liczbach i tekście
- Wprowadzanie danych
- Sterowanie przebiegiem programu
- Prawda i Fałsz

- Pętle
- Kolekcje - listy, kroki, słowniki
- Operator zakresu
- Wyrażenia listowe
- Funkcje, klasy
- Testy jednostkowe
- Przekazywanie argumentów do skryptu

AGILE DEVELOPMENT



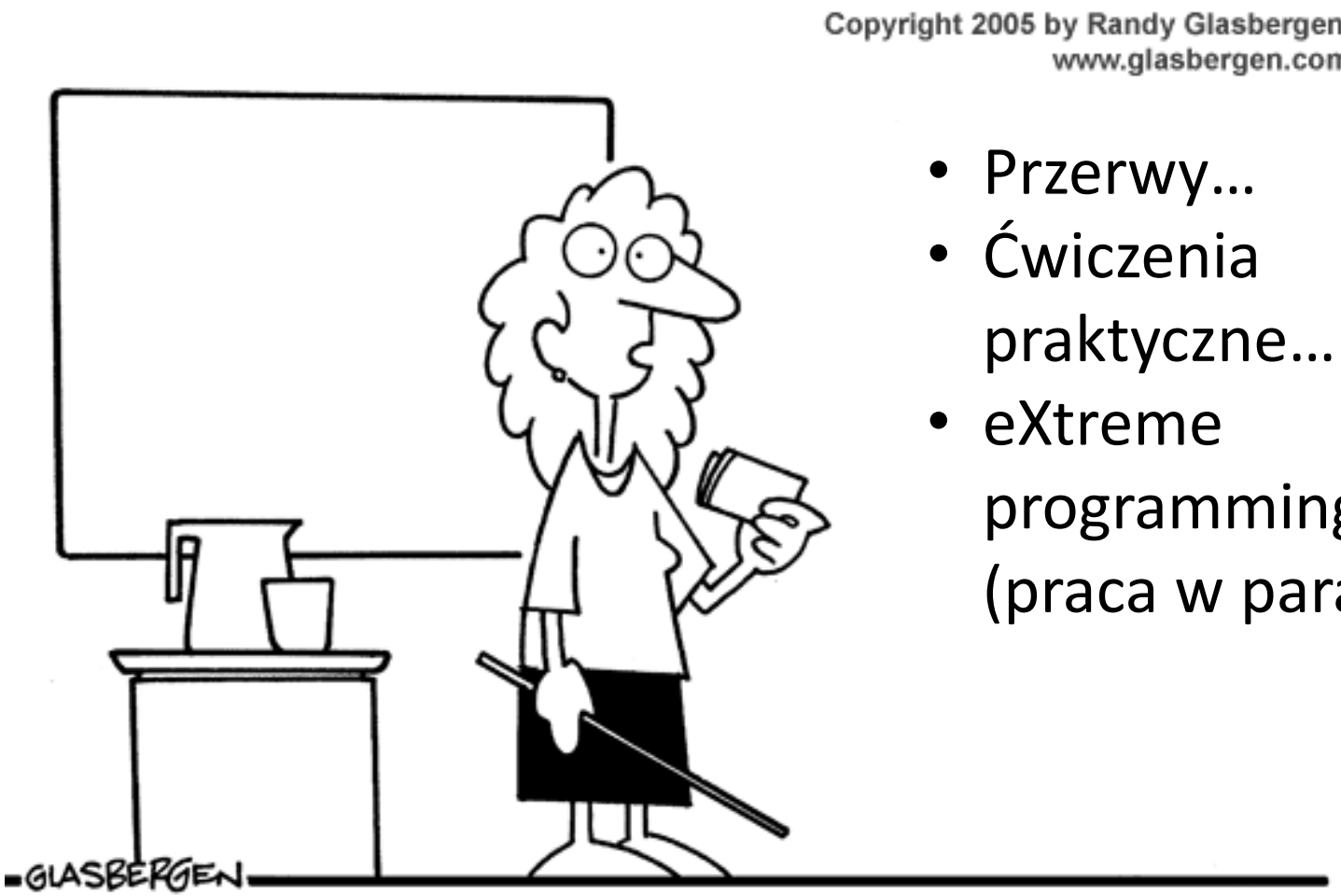
Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

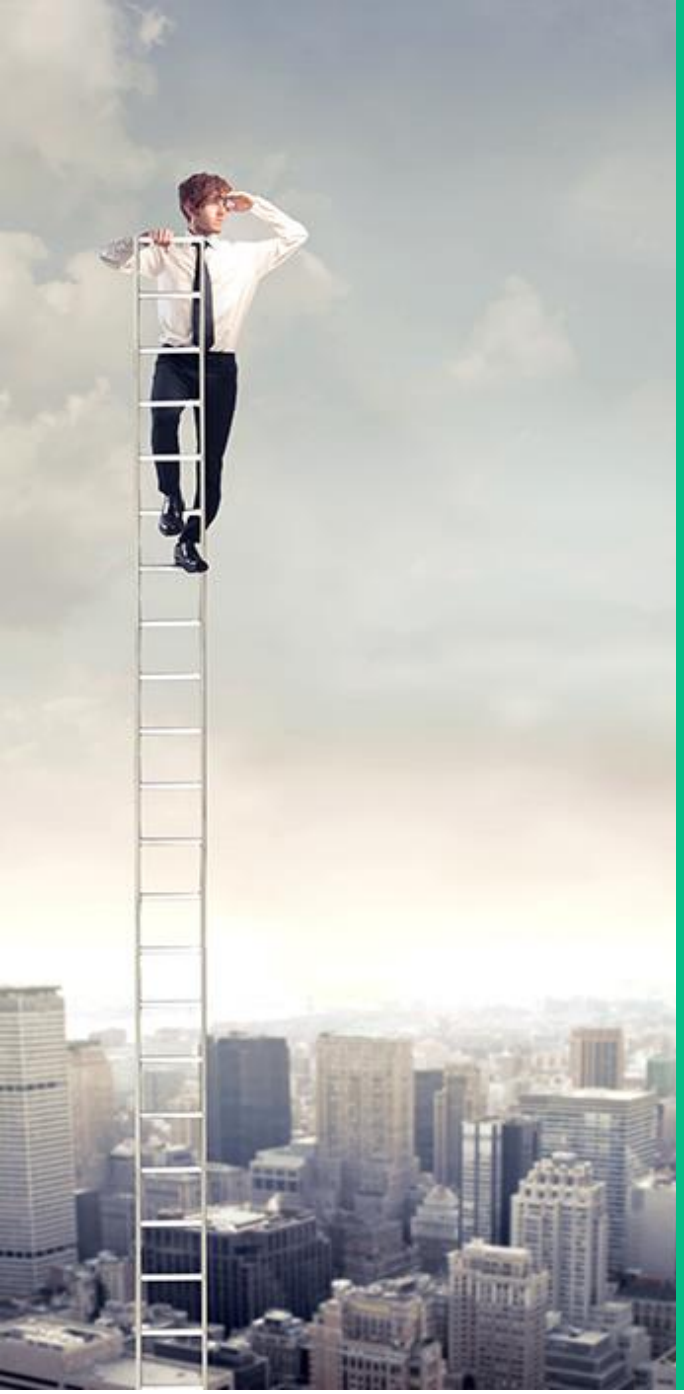
Responding to change over following a planern

USTALENIA TECHNICZNE



- Przerwy…
- Ćwiczenia praktyczne…
- eXtreme programming (praca w parach)

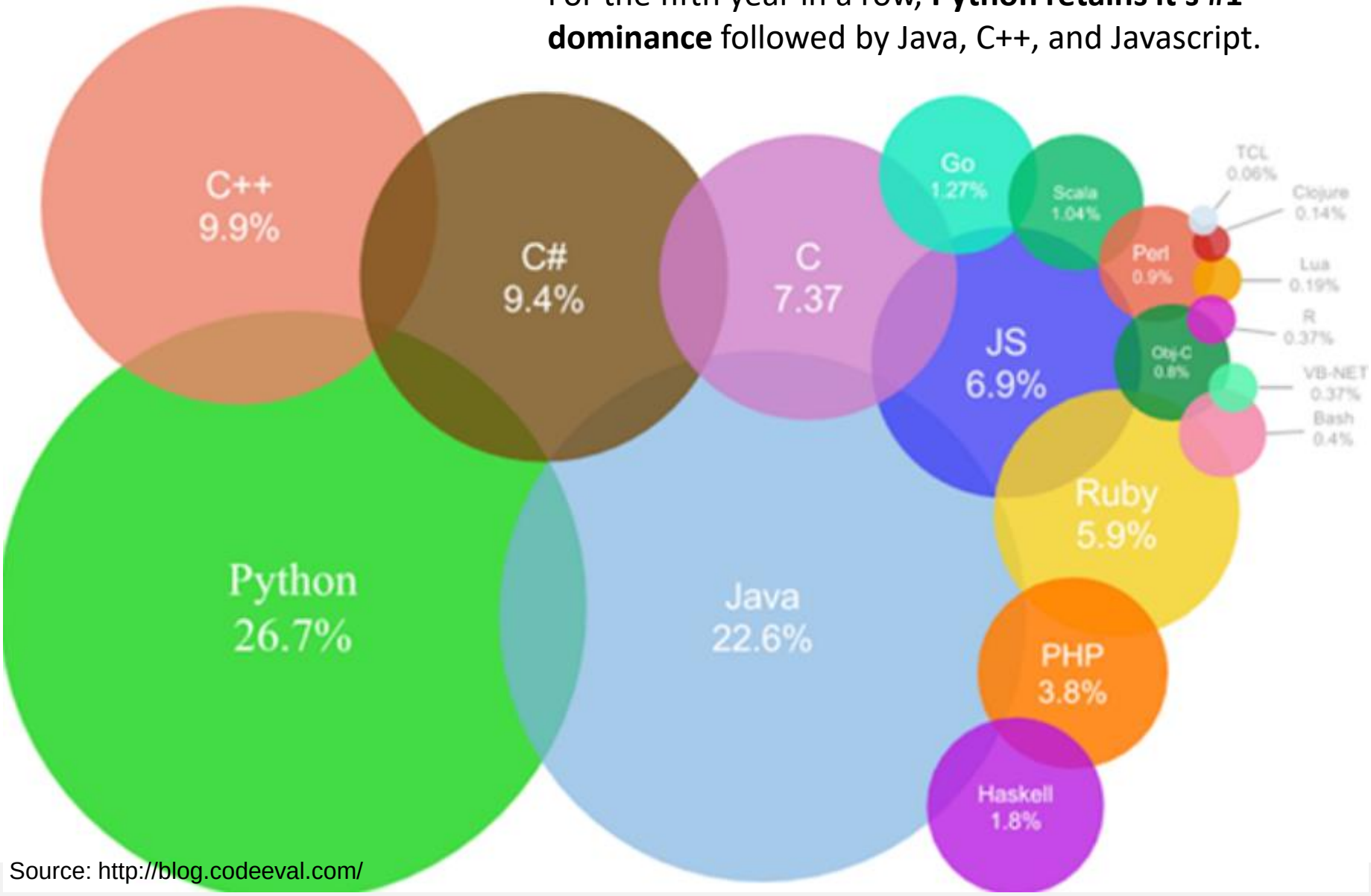
LEARNING BY DOING



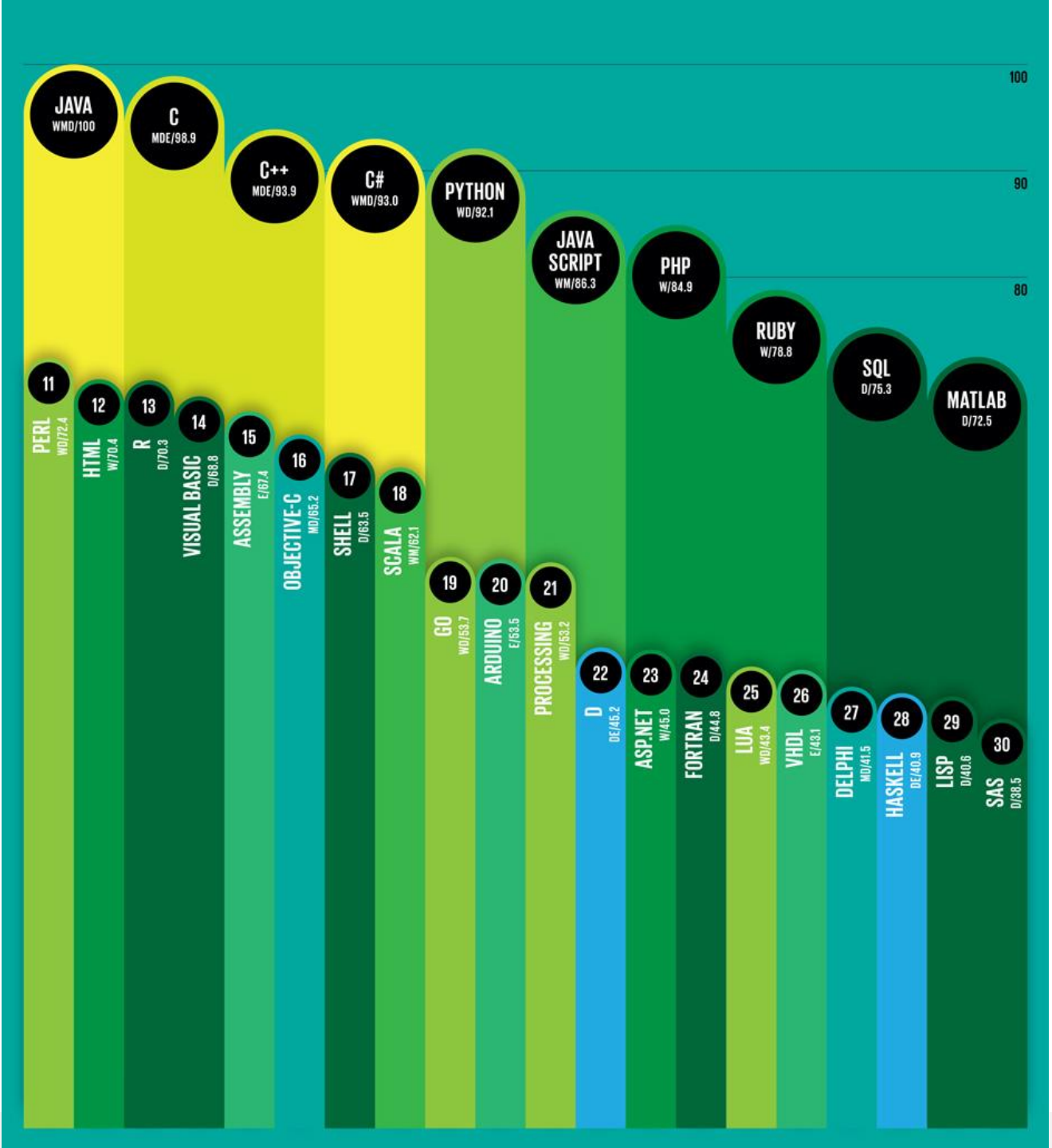
Założenie, filozofia i zastosowania Pytona

PROGRAMMING LANGUAGE POPULARITY

For the fifth year in a row, **Python retains it's #1 dominance** followed by Java, C++, and Javascript.



Source: http://blog.codeeval.com/



Working with computational journalist Nick Diakopoulos, IEEE Spectrum has weighted and combined 12 metrics from 10 sources (including IEEE Xplore, Google, and GitHub) to rank the most popular programming languages.

Source: http://spectrum.ieee.org/

Feb 2016	Feb 2015	Change	Programming Language	Ratings	Change
1	2	↑	Java	21.145%	+5.80%
2	1	↓	C	15.594%	-0.89%
3	3		C++	6.907%	+0.29%
4	5	↑	C#	4.400%	-1.34%
5	8	↑	Python	4.180%	+1.30%
6	7	↑	PHP	2.770%	-0.40%
7	9	↑	Visual Basic .NET	2.454%	+0.43%
8	12	⇈	Perl	2.251%	+0.86%
9	6	↓	JavaScript	2.201%	-1.31%
10	11	↑	Delphi/Object Pascal	2.163%	+0.59%
11	20	⇈	Ruby	2.053%	+1.18%
12	10	↓	Visual Basic	1.855%	+0.14%
13	26	⇈	Assembly language	1.828%	+1.08%
14	4	⇊	Objective-C	1.403%	-4.62%
15	30	⇈	D	1.391%	+0.77%
16	27	⇈	Swift	1.375%	+0.65%
17	18	↑	R	1.192%	+0.23%
18	17	↓	MATLAB		
19	13	⇊	PL/SQL		
20	33	⇈	Groovy		

Source: http://www.tiobe.com/tiobe_index

TIOBE
the software quality company

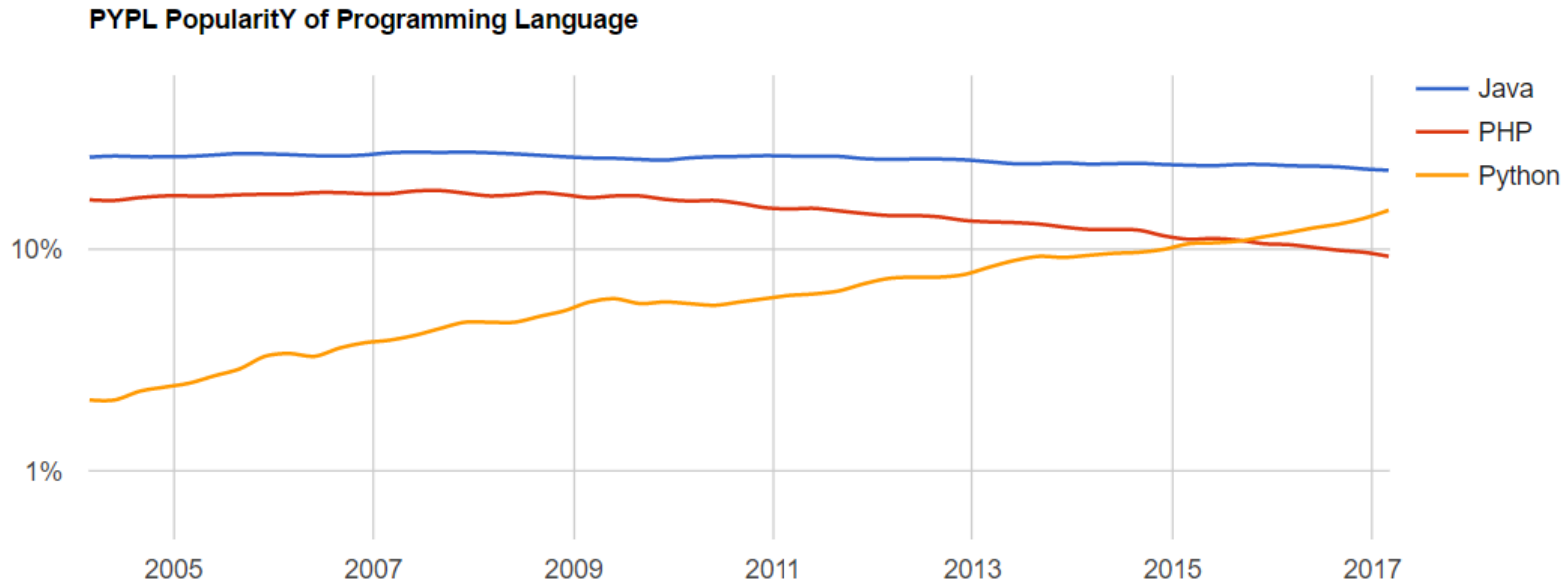
MOTIVATION

Worldwide, Mar 2017 compared to a year ago:

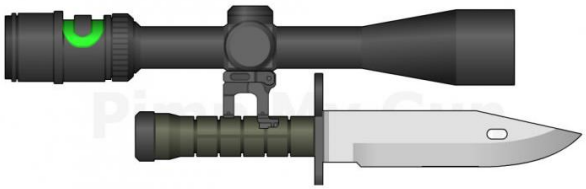
Rank	Language	Share	Trend
1	Java	22.7 %	-1.4 %
2	Python	15.0 %	+3.0 %
3	PHP	9.3 %	-1.2 %
4	C#	8.3 %	-0.4 %
5	Javascript	7.7 %	+0.4 %
6	C++	6.9 %	-0.5 %
7	C	6.9 %	-0.1 %

Source: The PYPL PopularitY of Programming Language Index is created by analyzing how often language tutorials are searched on Google.

Worldwide, Java is the most popular language, Python grew the most in the last 5 years (7.6%) and PHP lost the most (-4.9%)



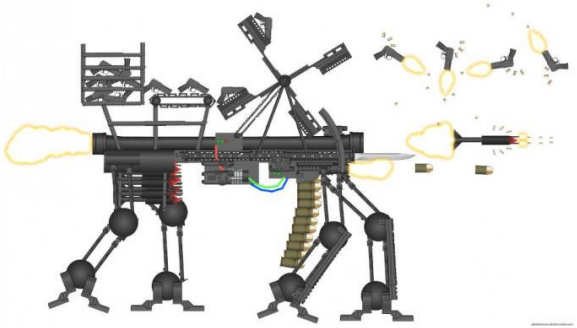
Assembly



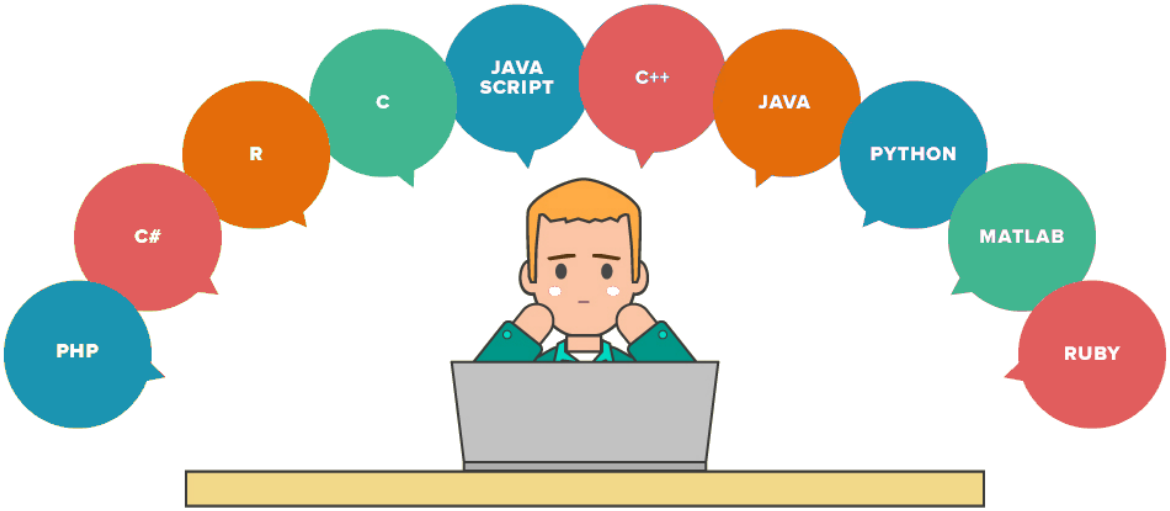
C

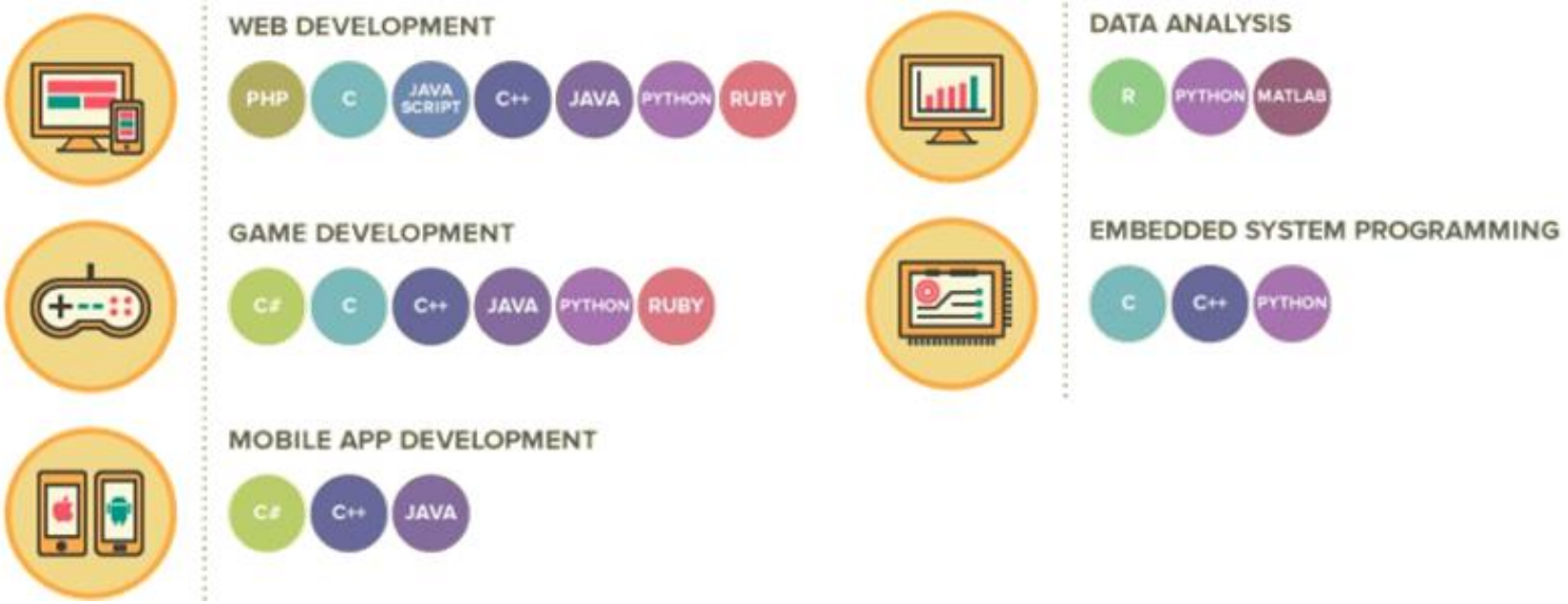
C++

Python



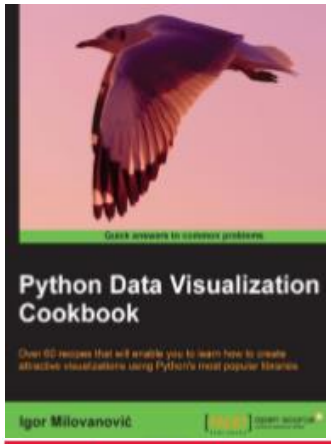
WHICH PROGRAMMING LANGUAGE PICK?

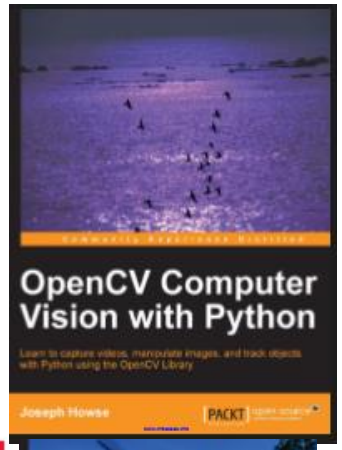


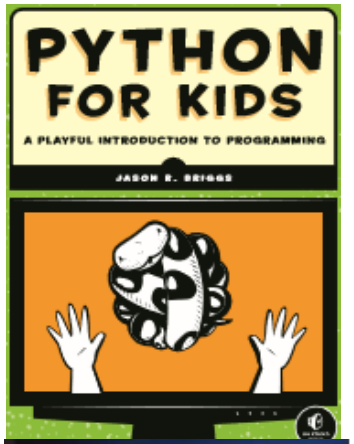


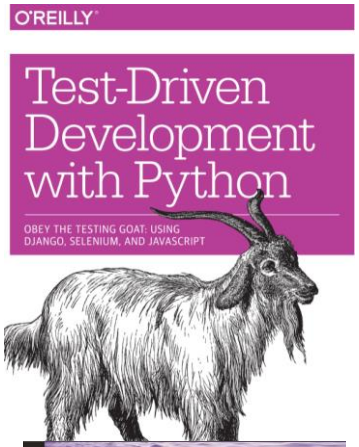
Source: http://blog.udacity.com/2015/05/pick-your-first-programming-language.html

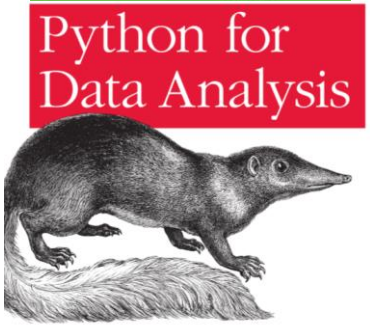
EXAMPLE OF PYTHON USAGE

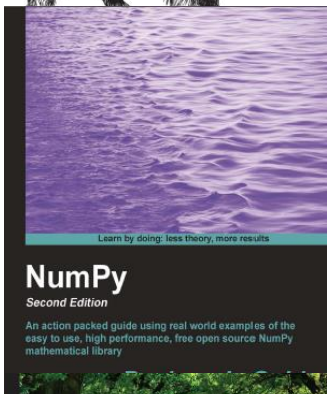


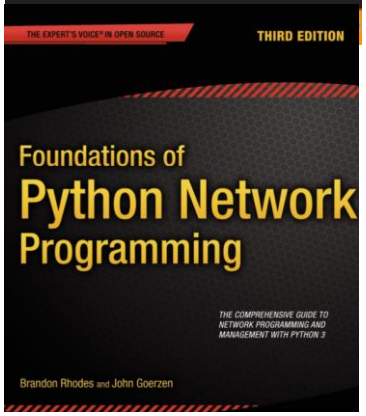


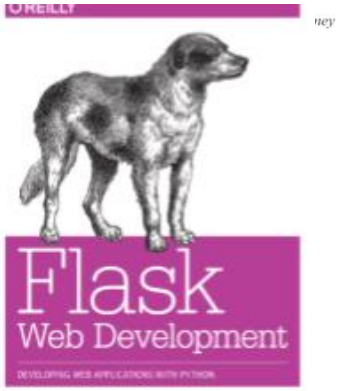


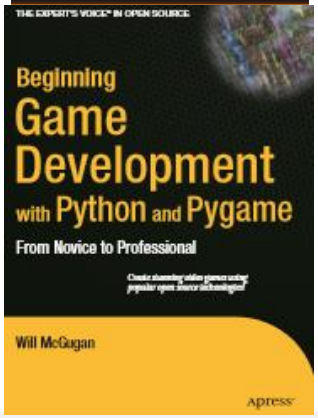


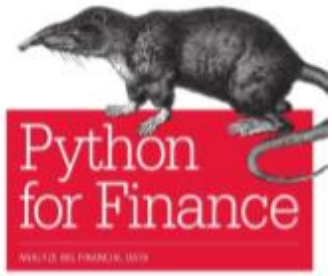


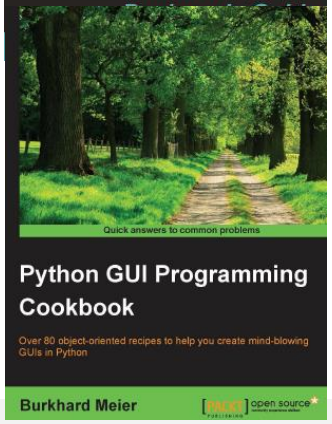




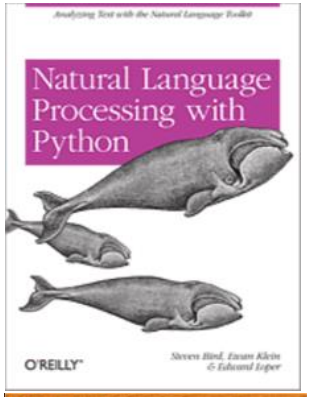


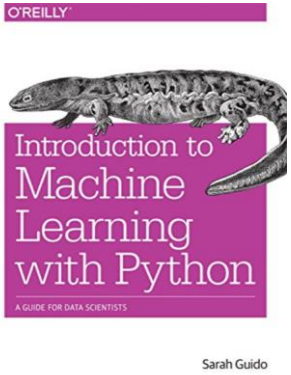


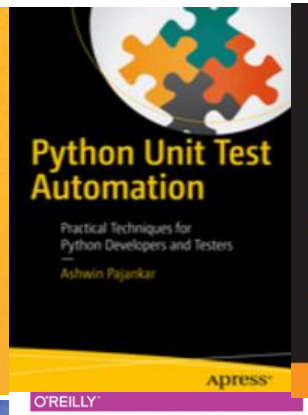


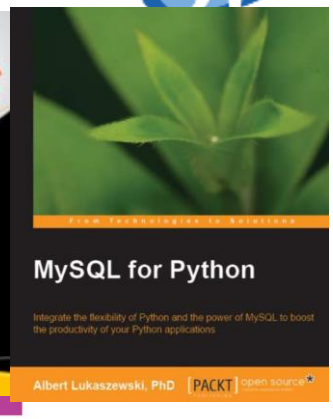


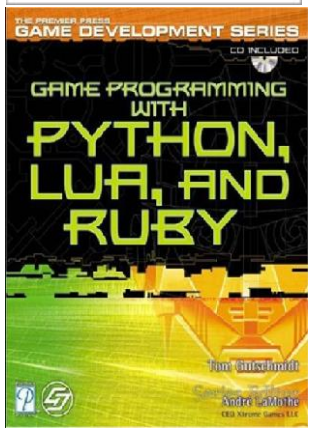
EXAMPLE OF PYTHON USAGE

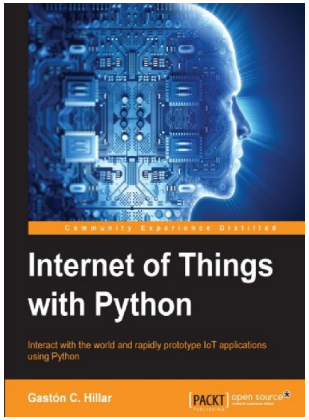


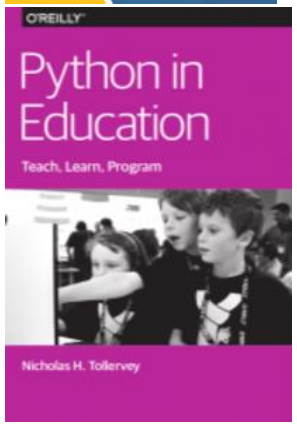


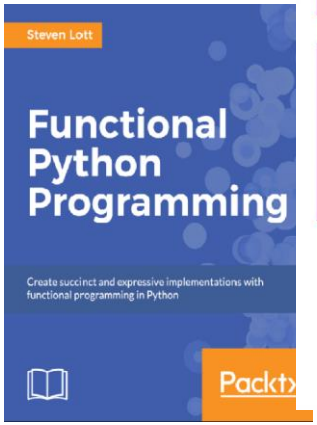


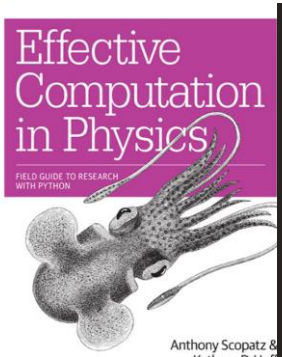


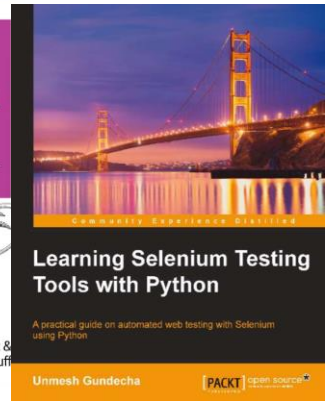


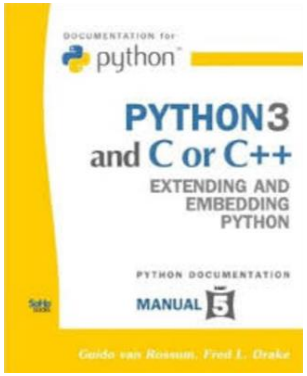


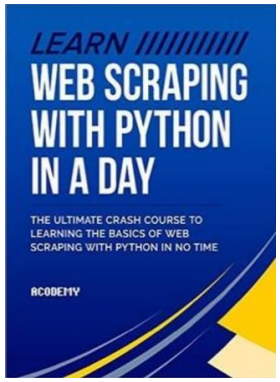


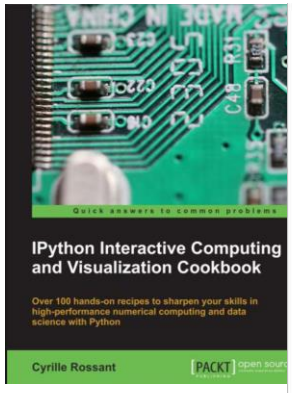


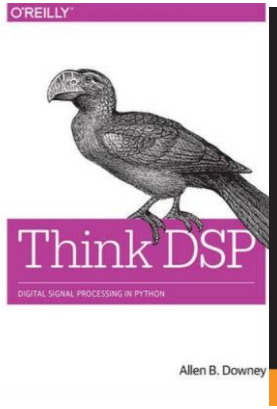












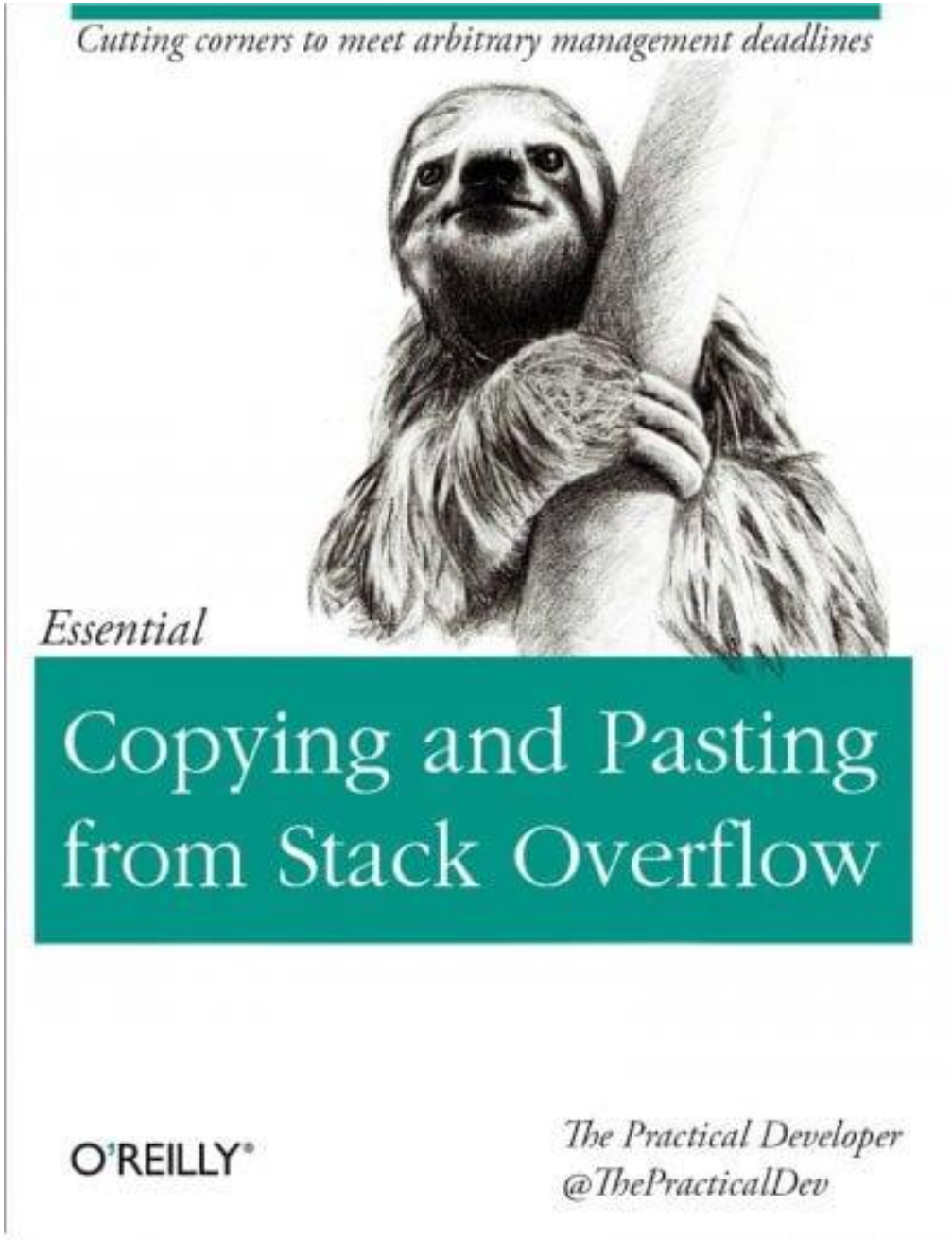
How to actually learn any new programming concept

Essential

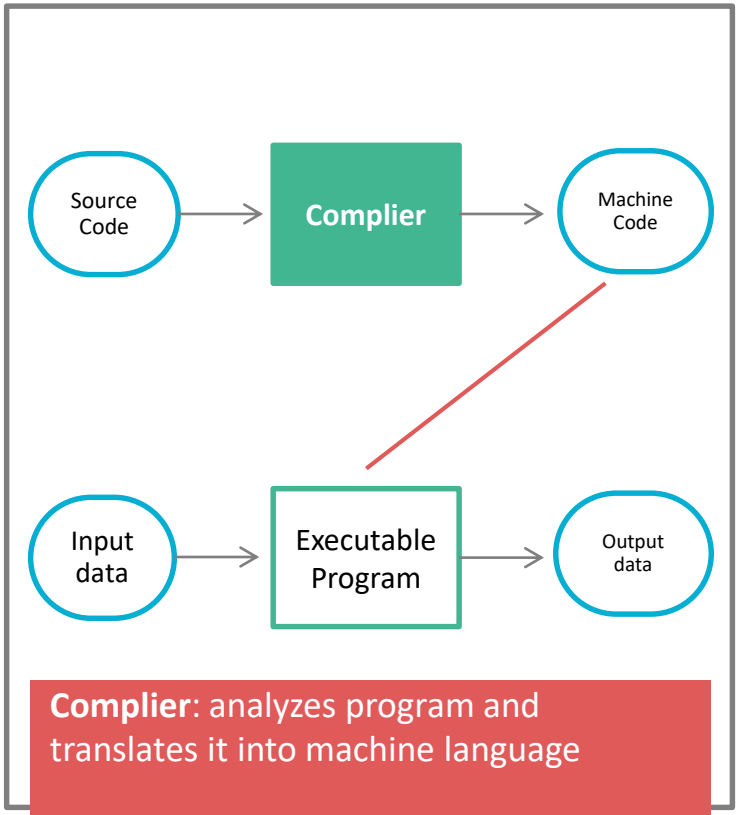
Changing Stuff and Seeing What Happens

O RLY?

@ThePracticalDev

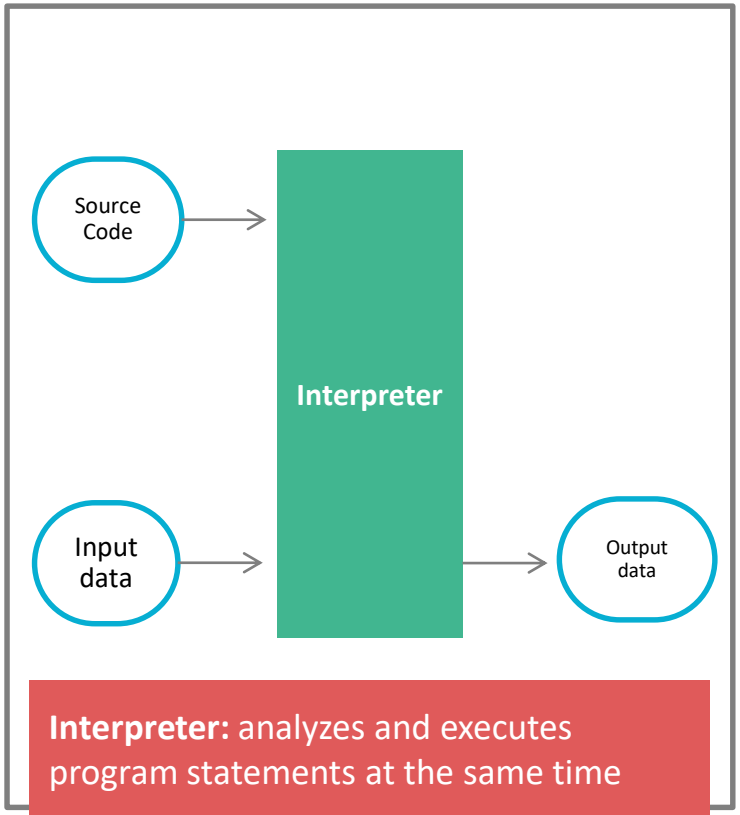

Cutting corners to meet arbitrary management deadlines
Essential
Copying and Pasting from Stack Overflow
O'REILLY
The Practical Developer
@ThePracticalDev

Compliers/Interpreters



Complier: analyzes program and translates it into machine language

Executable program: can be run independently from complier as many times → fast execution



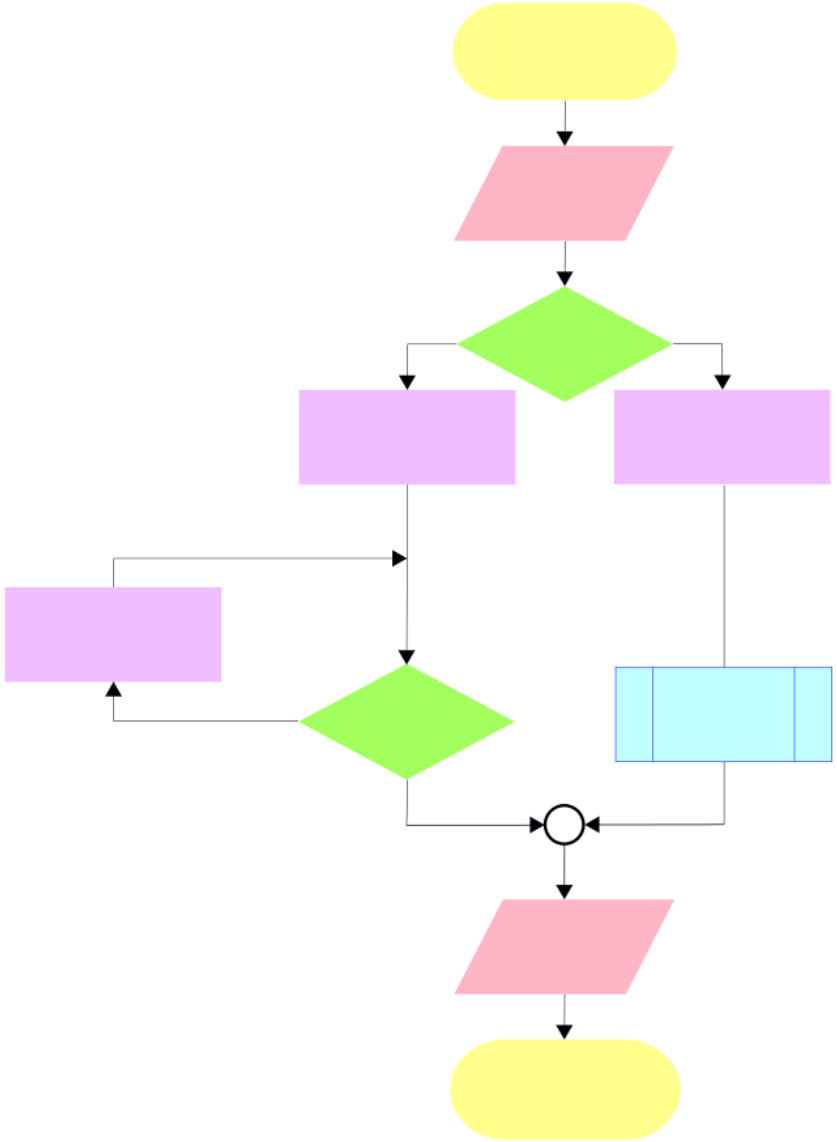
Interpreter: analyzes and executes program statements at the same time

Execution is slower
Easier to debug program

COMPILED VS INTERPRETED

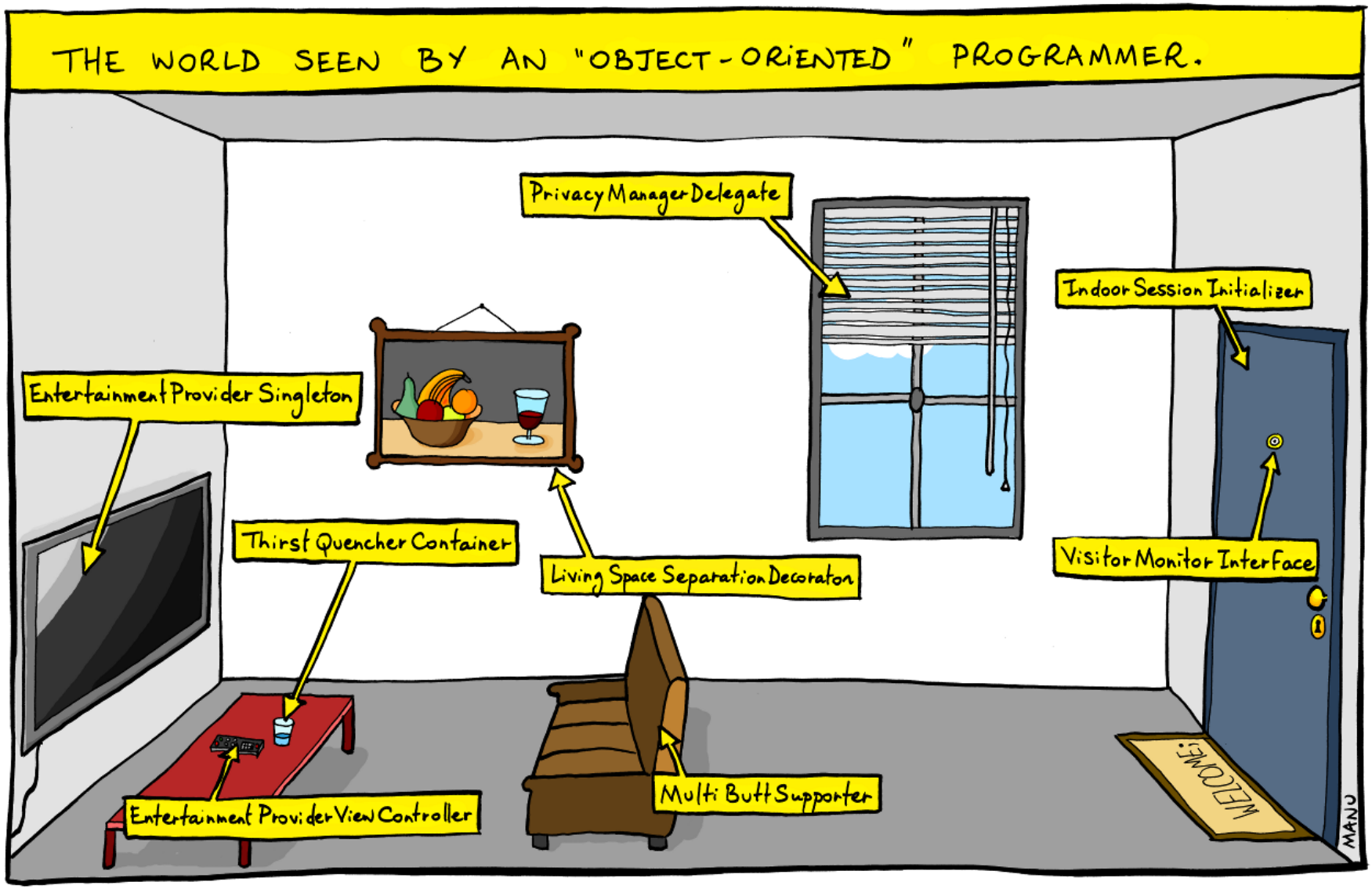
Compiled		Interpreted	
PROS	CONS	PROS	CONS
ready to run	**not** cross platform	cross-platform	interpreter required
Often **faster**	Inflexible	Simpler to test	often **slower**
source code is **private**	extra step	easier to debug	source code is **public**

PROCEDURAL PROGRAMMING



Source: https://commons.wikimedia.org/wiki/File:Flowchart_procedural_programming.svg

OBJECT ORIENTED PROGRAMMING

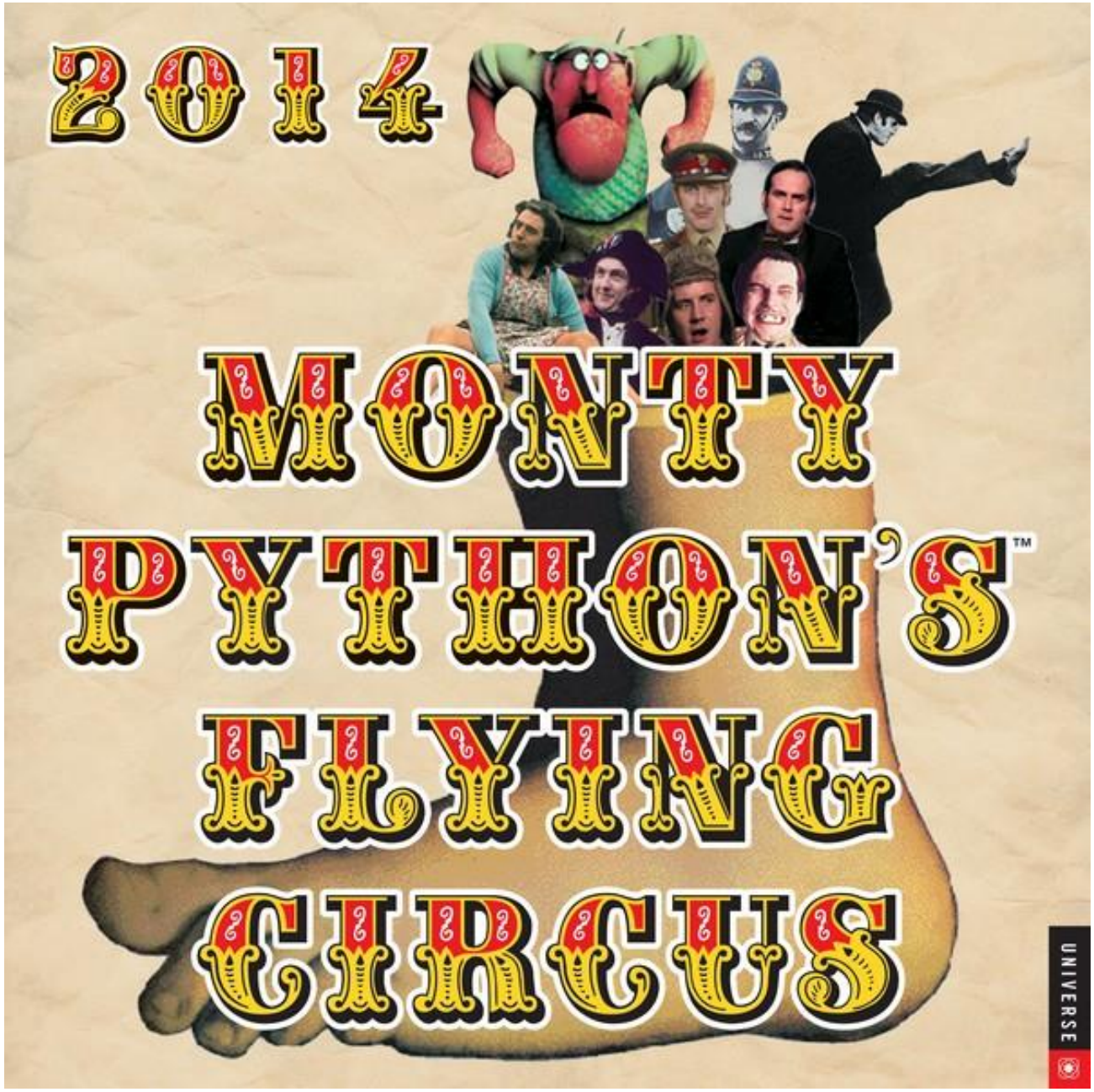


Source: http://www.bonkersworld.net/images/2011.09.07_object_oriented_programmer_world.png

PYTHON

Python is a widely used high-level, general-purpose, interpreted, dynamic programming language.
Its design philosophy emphasizes code **readability**, and its syntax allows programmers to express concepts in fewer lines of code than possible in languages such as C++ or Java.
The language provides constructs intended to enable clear programs on both a small and large scale.

INTRODUCTION TO PYTHON


2014
MONTY
PYTHON'S™
FLYING
CIRCUS
UNIVERSE

FREEDOM

"Python is an experiment in how much freedom programmers need. Too much freedom and nobody can read another's code; too little and expressiveness is endangered."

- Guido van Rossum

PYTHON – USAGE EXAMPLE

Use Python for...

Web Programming: Django, Pyramid, Bottle, Tornado, Flask, web2py

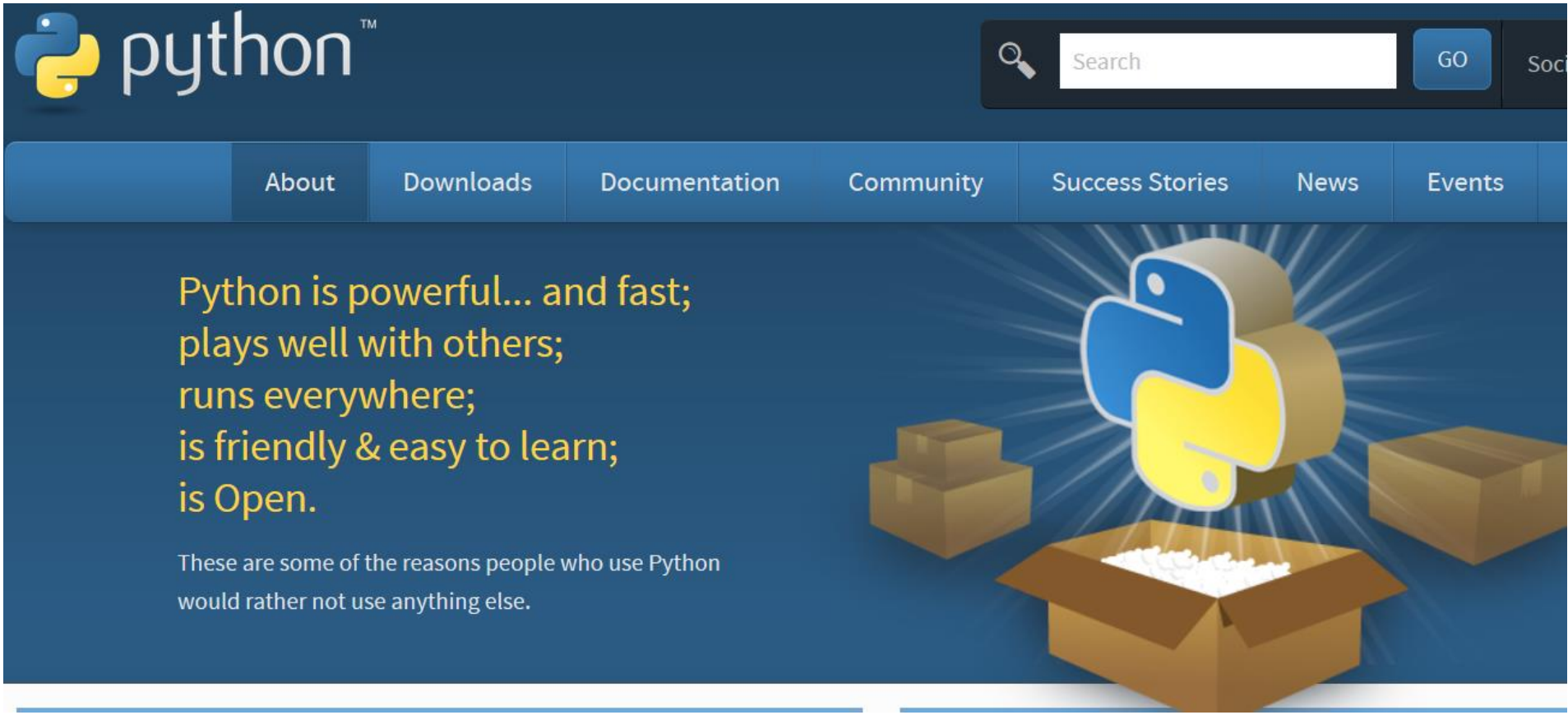
GUI Development: wxPython, tkInter, PyGtk, PyGObject, PyQt

Scientific and Numeric: SciPy, Pandas, IPython

Software Development: Buildbot, Trac, Roundup

System Administration: Ansible, Salt, OpenStack

python.org


python™
Search
GO
About
Downloads
Documentation
Community
Success Stories
News
Events
Python is powerful... and fast;
plays well with others;
runs everywhere;
is friendly & easy to learn;
is Open.
These are some of the reasons people who use Python
would rather not use anything else.

2.X VS 3.X

- Python’s creator (Guido van Rossum) and others decided to bundle the hard fixes together and call it Python 3.
- Python 2 is the **past**, and Python 3 is the **future**.
- The last version of Python 2 is 2.7, and it will be supported for a long time, but it’s the end of the line; **there will be no Python 2.8**.
- New development will be in Python 3.

Source: "Introducing Python. Modern Computing In Simple Packages", B. Lubanovic

2.X VS 3.X

Python 2.7 will retire in...

Python 2.7 will not be maintained past 2020.

My original idea was to throw a Python 2 Celebration of Life party at PyCon 2020, to celebrate everything Python 2 did for us. That idea still stands. (If this sounds interesting to you,
email pythonclockorg@gmail.com).

Python 2, thank you for your years of faithful service.
Python 3, your time is now

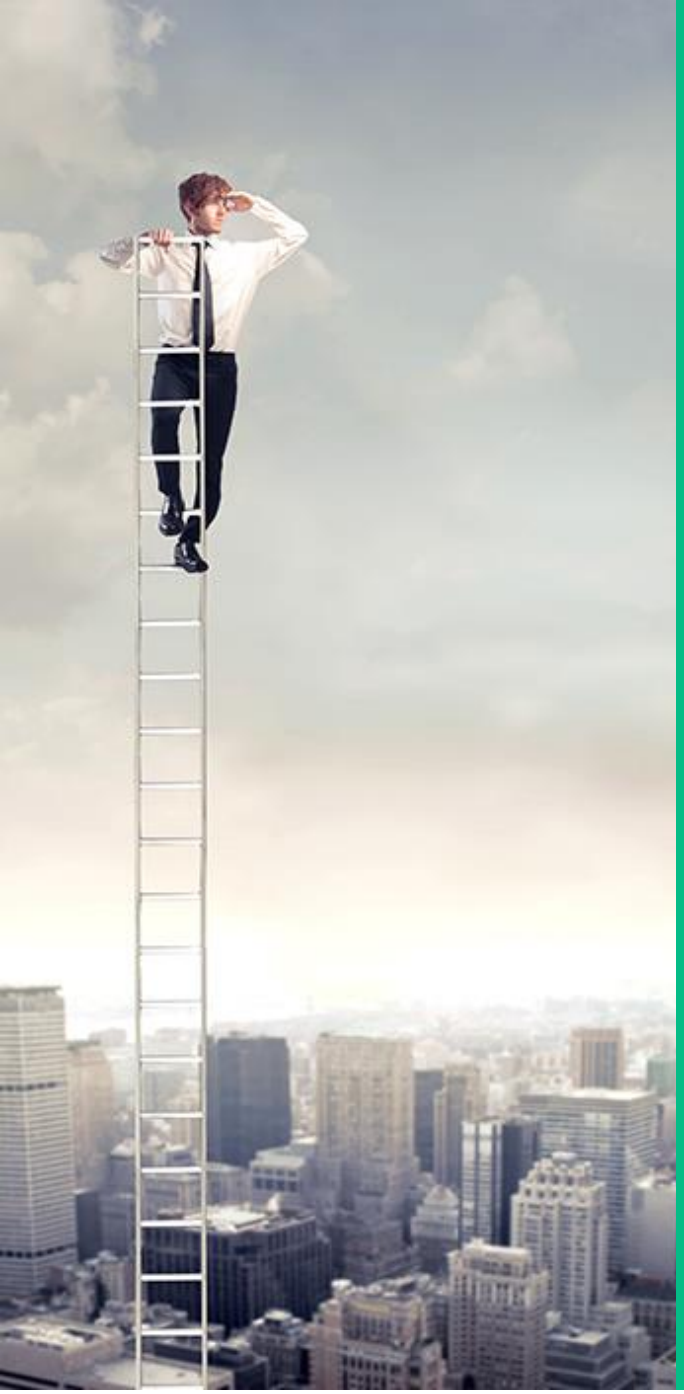
Source: https://pythonclock.org/

PYTHON INTERPRETER TYPES

The following implementations may be comprehensive or even complete, but at the very least can be said to be working in that you can run typical programs with them already:

- Brython - a way to run Python in the browser through translation to JavaScript
- CLPython - Python in Common Lisp
- HotPy - a virtual machine for Python supporting bytecode optimisation and translation (to native code) using type information gathered at run-time
- IronPython - Python in C# for the Common Language Runtime (CLR/.NET) and the FePy project's IronPython Community Edition (IPCE)
- Jython - Python in Java for the Java platform
- pyjs - (formally Pyjamas) a Python to JavaScript compiler plus Web/GUI framework
- PyMite - Python for embedded devices
- PyPy - Python in Python, targeting several environments
- pyvm - a Python-related virtual machine and software suite providing a nearly self-contained "userspace" system
- RapydScript - a Python-like language that compiles to JavaScript
- SNAPpy - "a subset of the Python language that has been optimized for use in low-power embedded devices" (apparently proprietary)
- tinypy - a minimalist implementation of Python in 64K of code

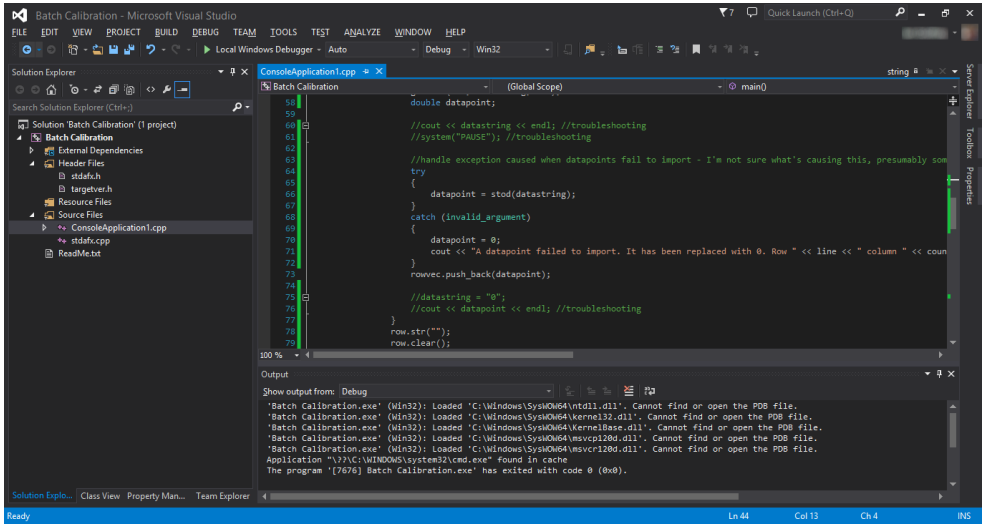
Source: https://wiki.python.org/moin/PythonImplementations



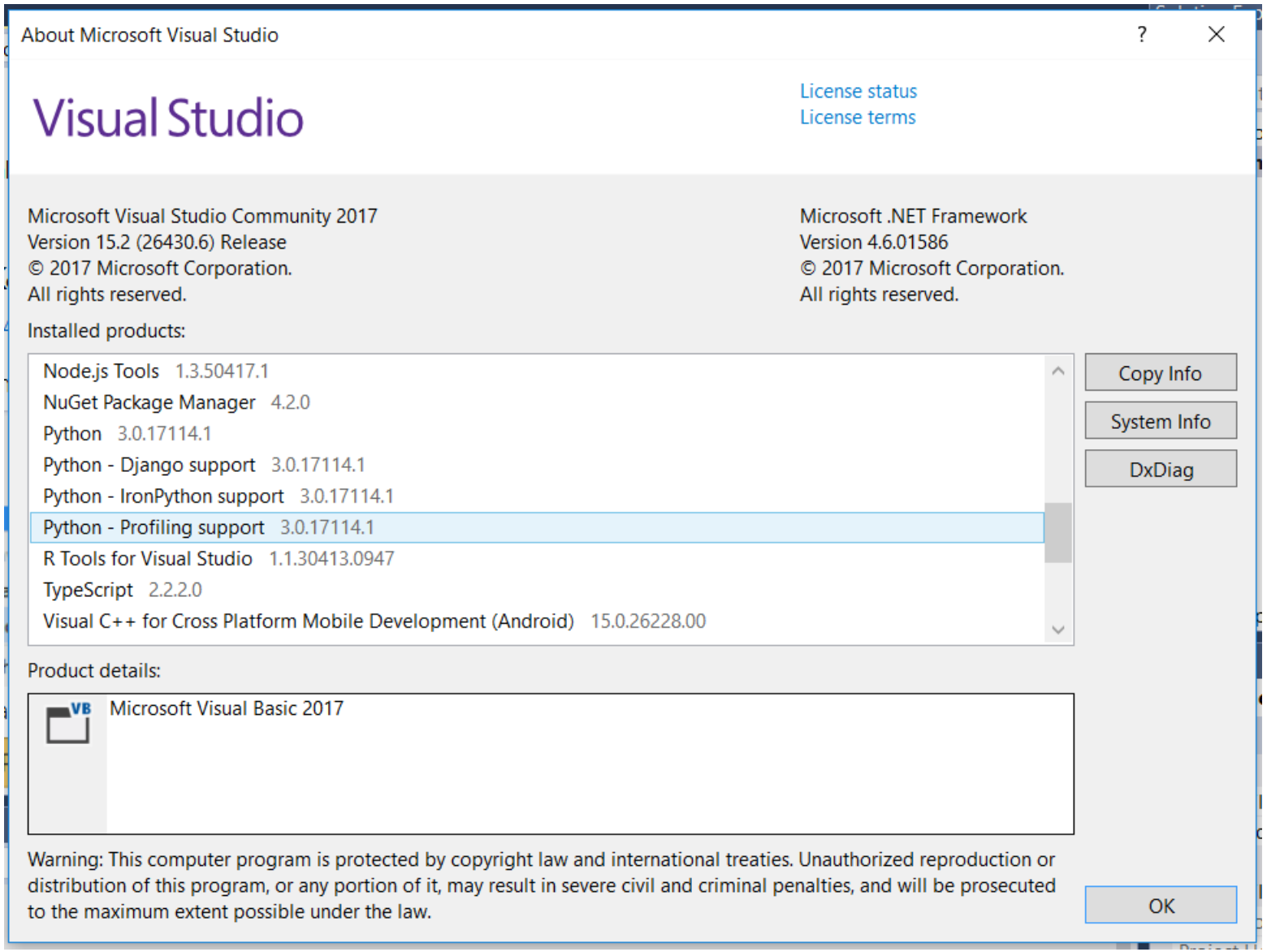
Tools for developers

TOOLS FOR DEVELOPERS

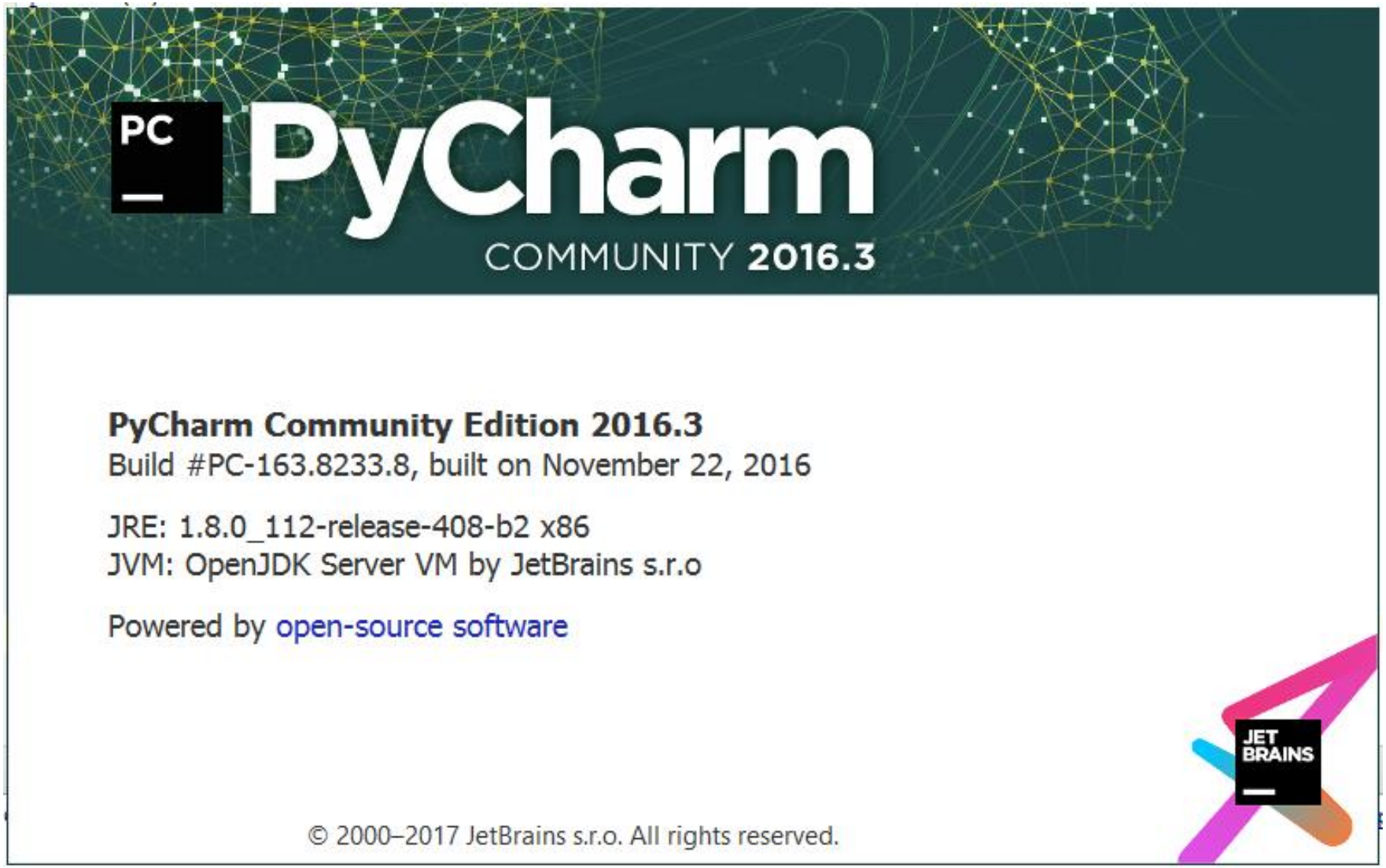
An **integrated development environment** (IDE) is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of a **source code editor**, **build automation tools** and a **debugger**. Most modern IDEs have intelligent code completion, a class browser, an object browser, and a class hierarchy diagram, for use in object-oriented software development.



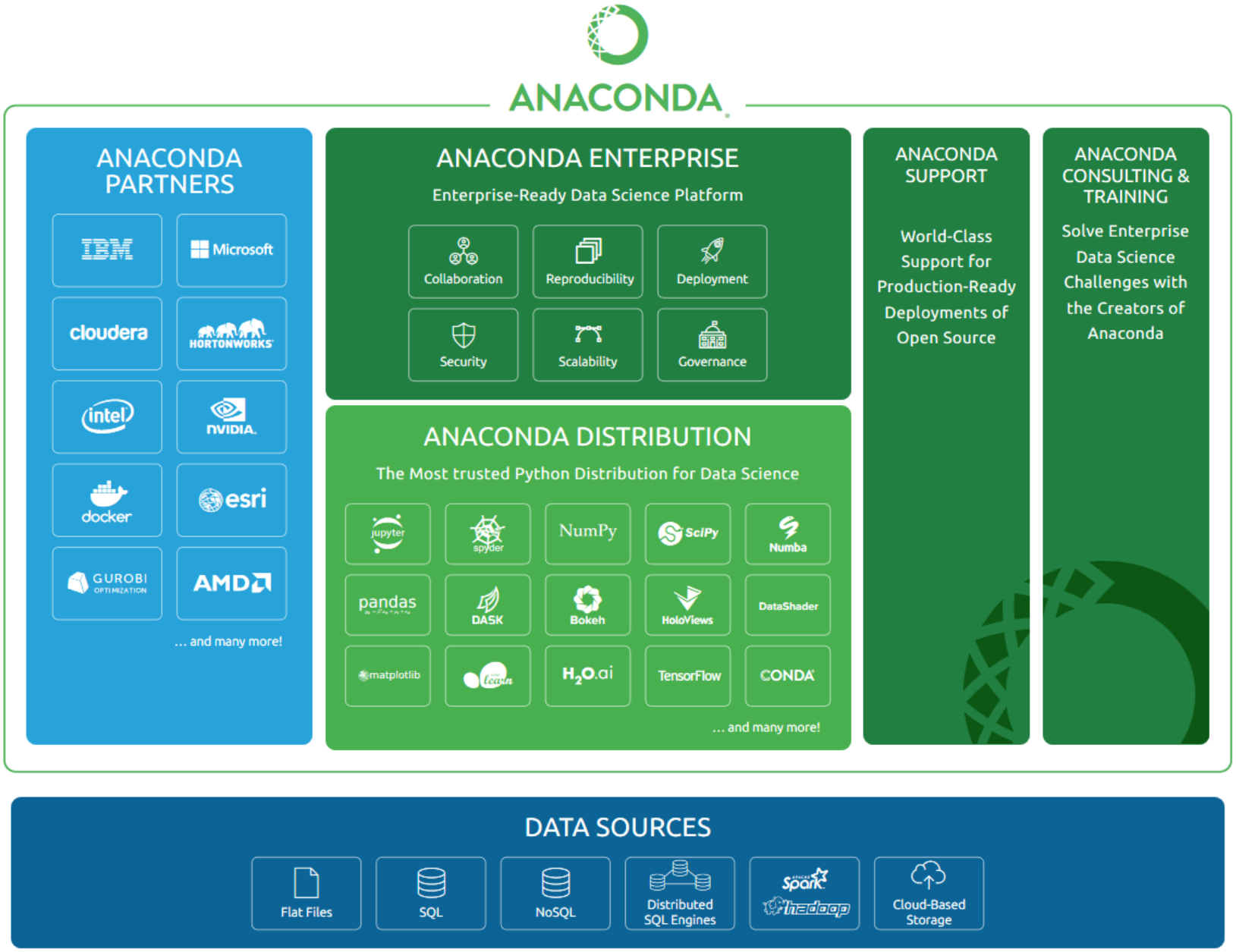
TOOLS FOR DEVELOPERS – MICROSOFT VISUAL STUDIO



TOOLS FOR DEVELOPERS – PYCHARM

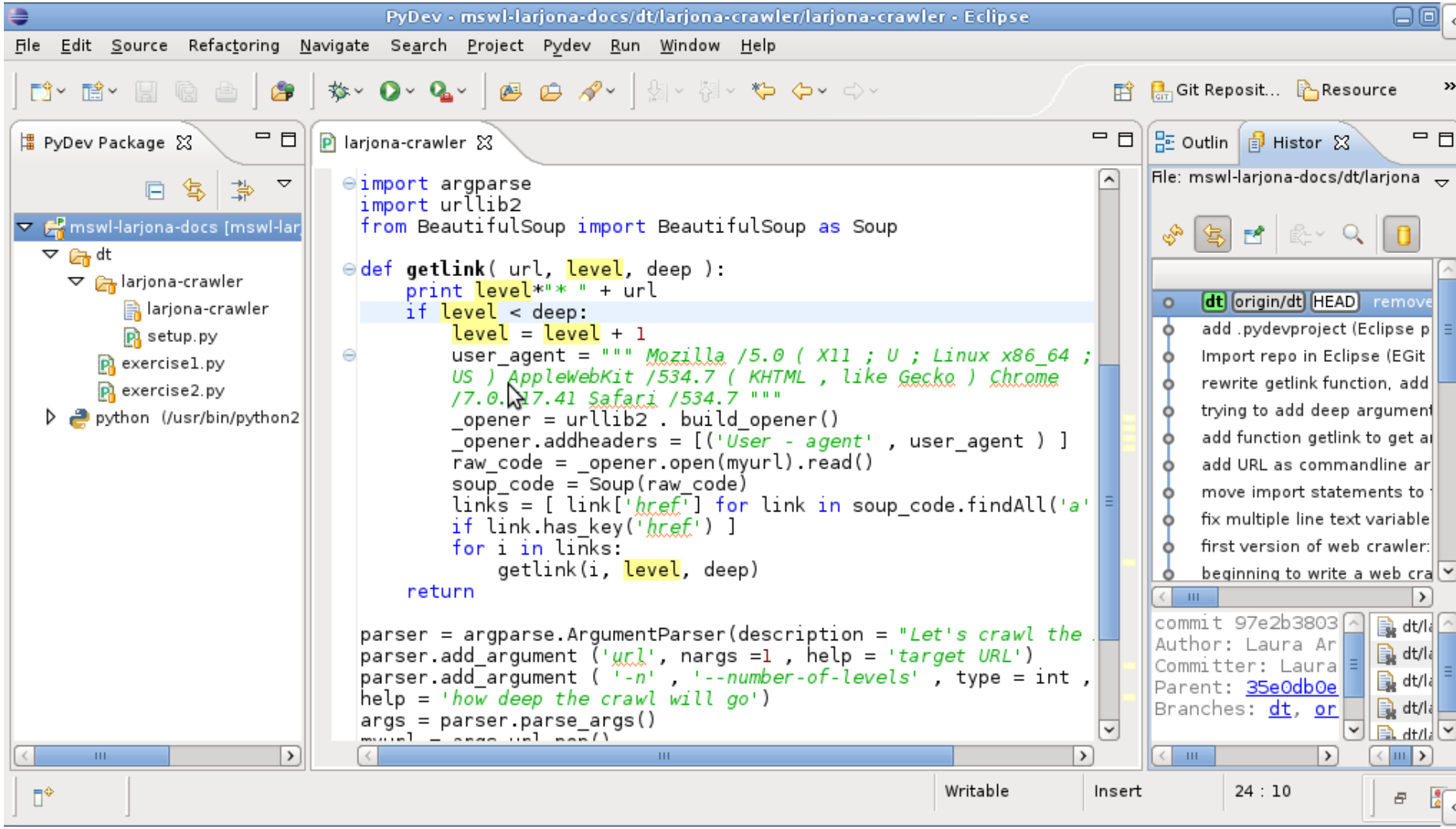


TOOLS FOR DEVELOPERS – ANACONDA



Source: https://www.continuum.io/what-is-anaconda

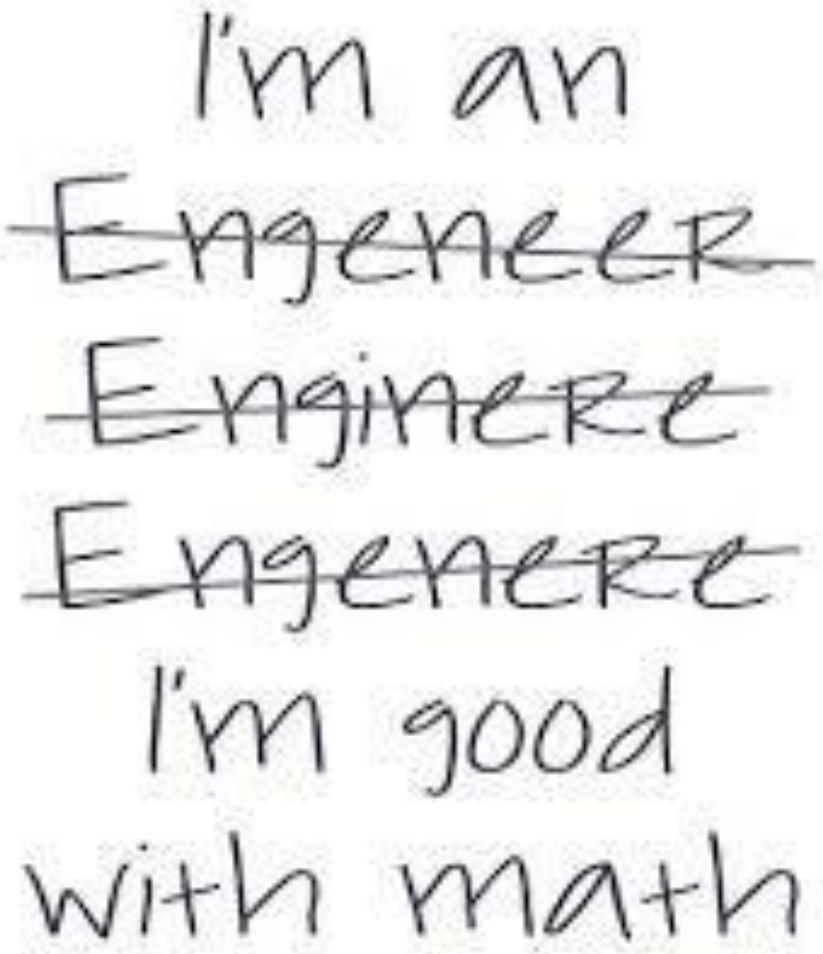
TOOLS FOR DEVELOPERS – PYDEV



https://marketplace.eclipse.org/content/pydev-python-ide-eclipse

Interacitve console:

>>>


I'm an
Engeneer
Enginere
Engenere
I'm good
with math

CALCULATOR

Operator	Description	Example	Result
+	addition	`5 + 8`	`13`
-	subtraction	`90 - 10`	`80`
*	multiplication	`4 * 7`	`28`
/	floating point division	`7 / 2`	`3.5`
//	integer (truncating) division	`7 // 2`	`3`
%	modulus (remainder)	`7 % 3`	`1`
**	exponentiation	`3 ** 4`	`81`

Source: "Introducing Python. Modern Computing In Simple Packages", B. Lubanovic

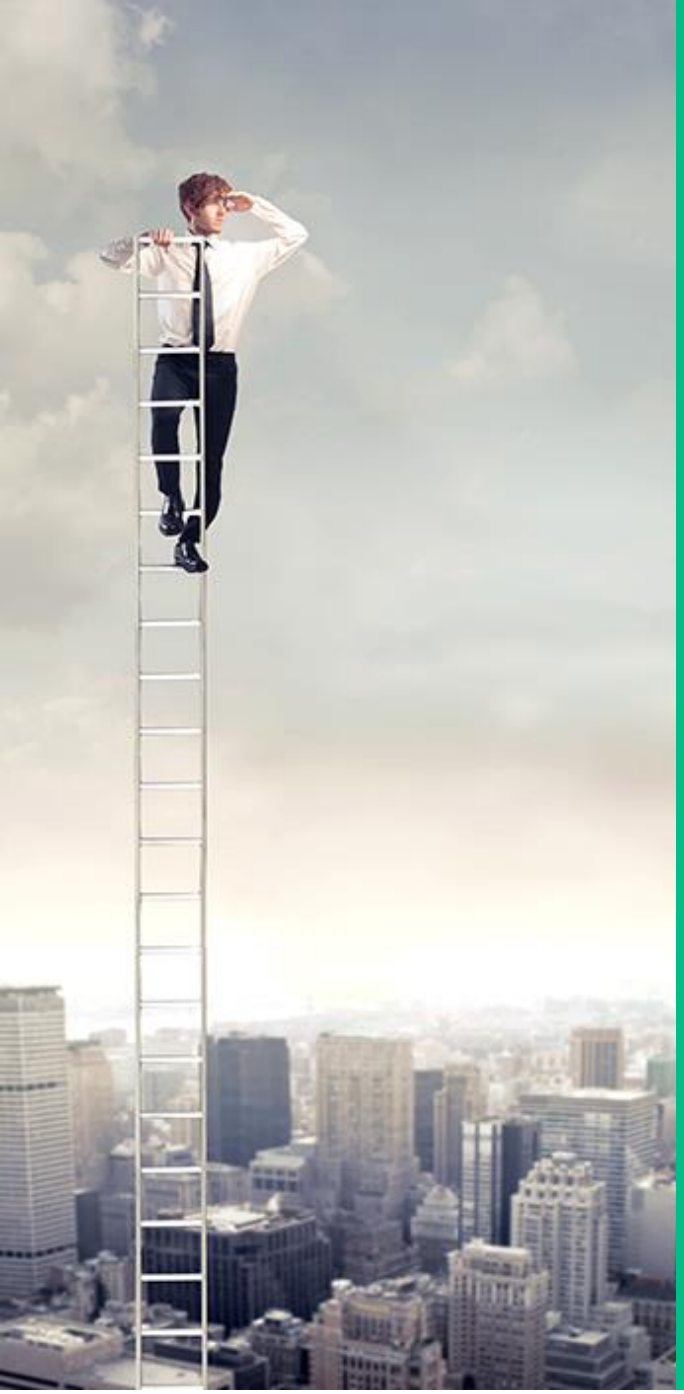
INTRODUCTION TO PYTHON

„Always look on the bright side of life”

CODE COMMENTS

```
# this is the first comment spam = 1
# and this is the second comment
# ... and now a third!


text = "# This is not a comment."
```


REAL Programmers
DON'T COMMENT
their CODE.
If it was
HARD TO WRITE
it should be
HARD
to UNDERSTAND
MemeCenter.com
memecenter.com
MemeCenter



Basic data types

Variables example

```
people = 12
tax = 12.5 / 100
price = 100.50
print (price*tax)
```

STRINGS

Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes (’...’) or double quotes ("...") with the same result

```
Text_variable = "my text example"
```

s = ’First line.\nSecond line.’

```
>>> spam = 'Say hi to Bob\'s mother.'
```

Escape Characters

Escape character	Prints as
\'	Single quote
\"	Double quote
\t	Tab
\n	Newline (line break)
\\	Backslash

INPUT

The syntax for input() is

```
input([prompt])
```

where prompt is the string we wish to display on the screen. It is optional.

```
>>> num = input('Enter a number: ')
Enter a number: 10
>>> num
'10'
```

CONVERSION

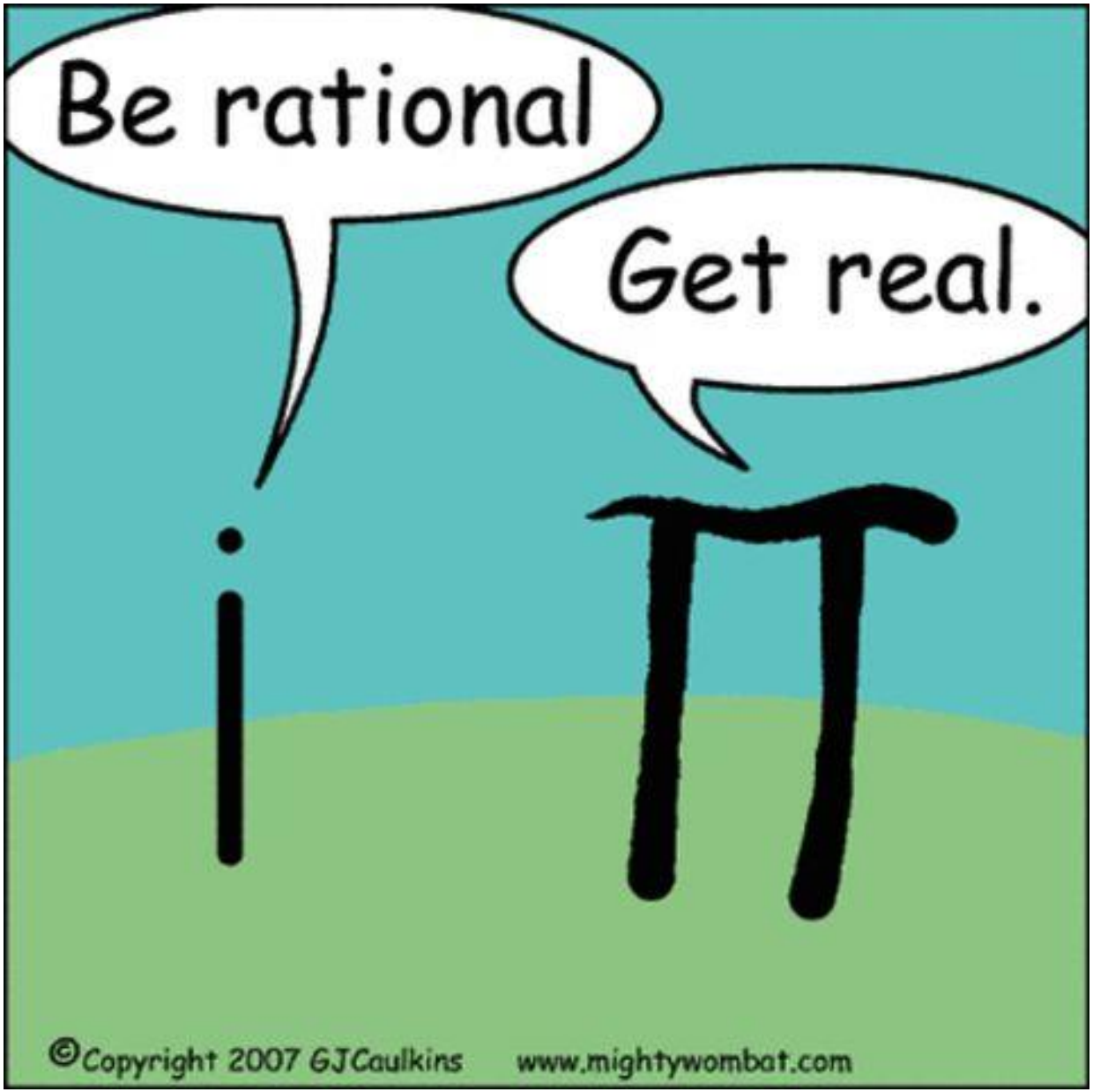
Here, we can see that the entered value 10 is a string, not a number. To convert this into a number we can use **int()** or **float()** functions.

```
>>> int('10')
10
>>> float('10')
10.0
```

BINARY, OCTAL, HEXADECIMAL

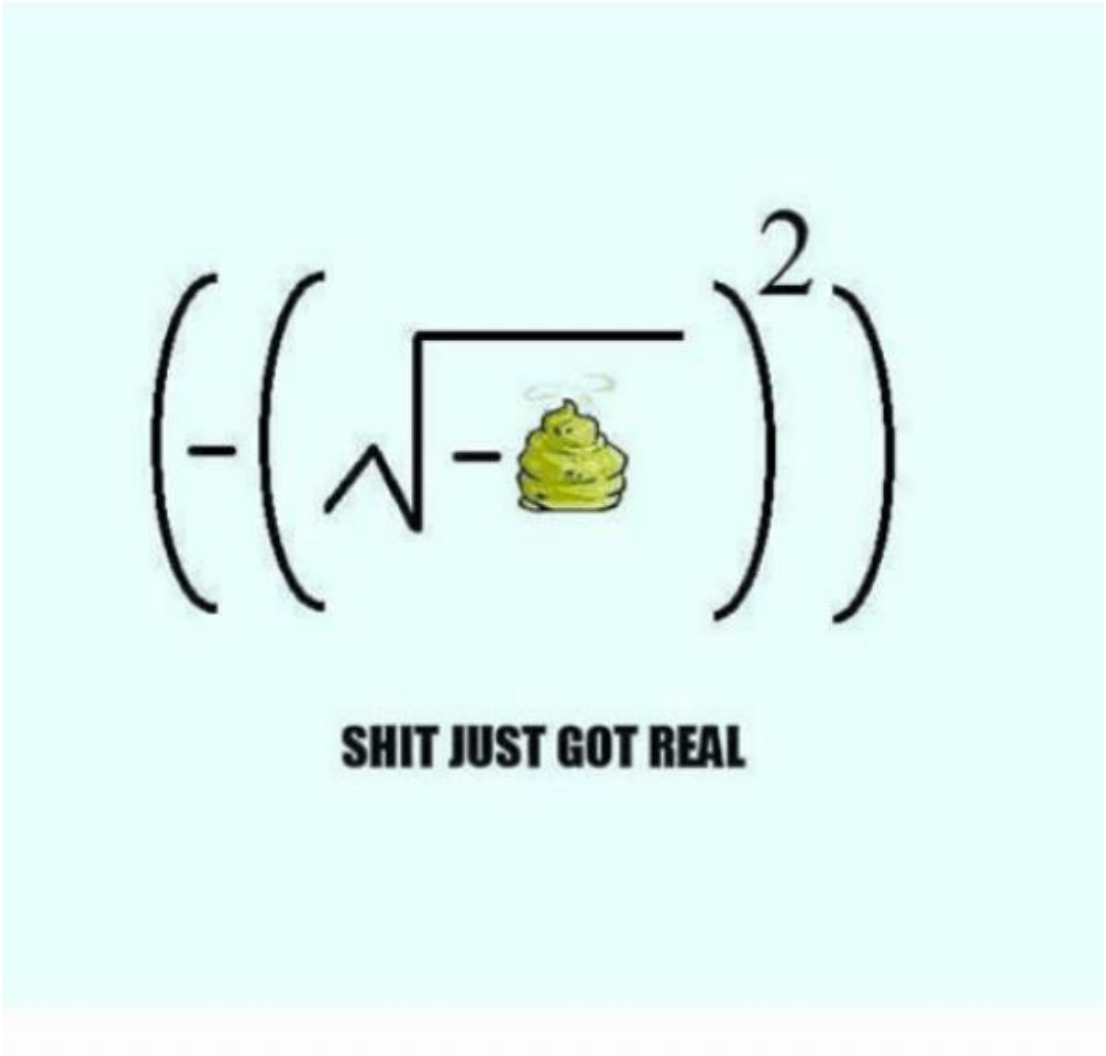
```
number = 0b10      #binary number (0, 1)
print (number)     #2

number = 0o10      #octal number (0 - 7)
print(number)      #8

number = 0x10      #hexadecimal numbers (0-9; A-F)
print(number)      #16


#bin()    - converts number to binary
#oct()    - converts number to octal
#hex()    - converts number to hexadecimal

print(bin(100))       #0b1100100
```

COMPLEX NUMBERS



COMPLEX NUMBERS

```
#example of complex numbers
x = 1 + 1j
y = 2 + 3j

print (x + y)
print (x * y)
```



#engineeringmemes #memes #funny #mechanicalengineering #civilengineering #engineering

CONCATENATE

If you want to concatenate variables or a variable and a literal, use +

```
text = "textA" + "textB"
```

INDEXING

```
word = "Python"
```

Indices may also be negative numbers, to start counting from the right:

```
word[-1]  # last character’n’
word[-2]  # second-last character’o’
word[-6]  #?
```

SLICING

In addition to indexing, slicing is also supported. While indexing is used to obtain individual characters, slicing allows you to obtain substring:

```
word[0:2]
# characters from position 0 (included) to 2
(excluded)
'Py'
 word[2:5]
# characters from position 2 (included) to 5
(excluded)
'tho'
```

Note how the start is always included, and the end always excluded.

STRINGS

Python strings cannot be changed — they are immutable. Therefore, assigning to an indexed position in the string results in an error:

word[0] = ’J’ #ERROR!

EMBEDDING VALUES IN STRINGS

If you want to display a message using the contents of a variable, you can embed values in a string using %s, which is like a marker for a value that you want to add later.

```
>>> myscore = 1000
>>> message = 'I scored %s points'
>>> print(message % myscore)
I scored 1000 points
```

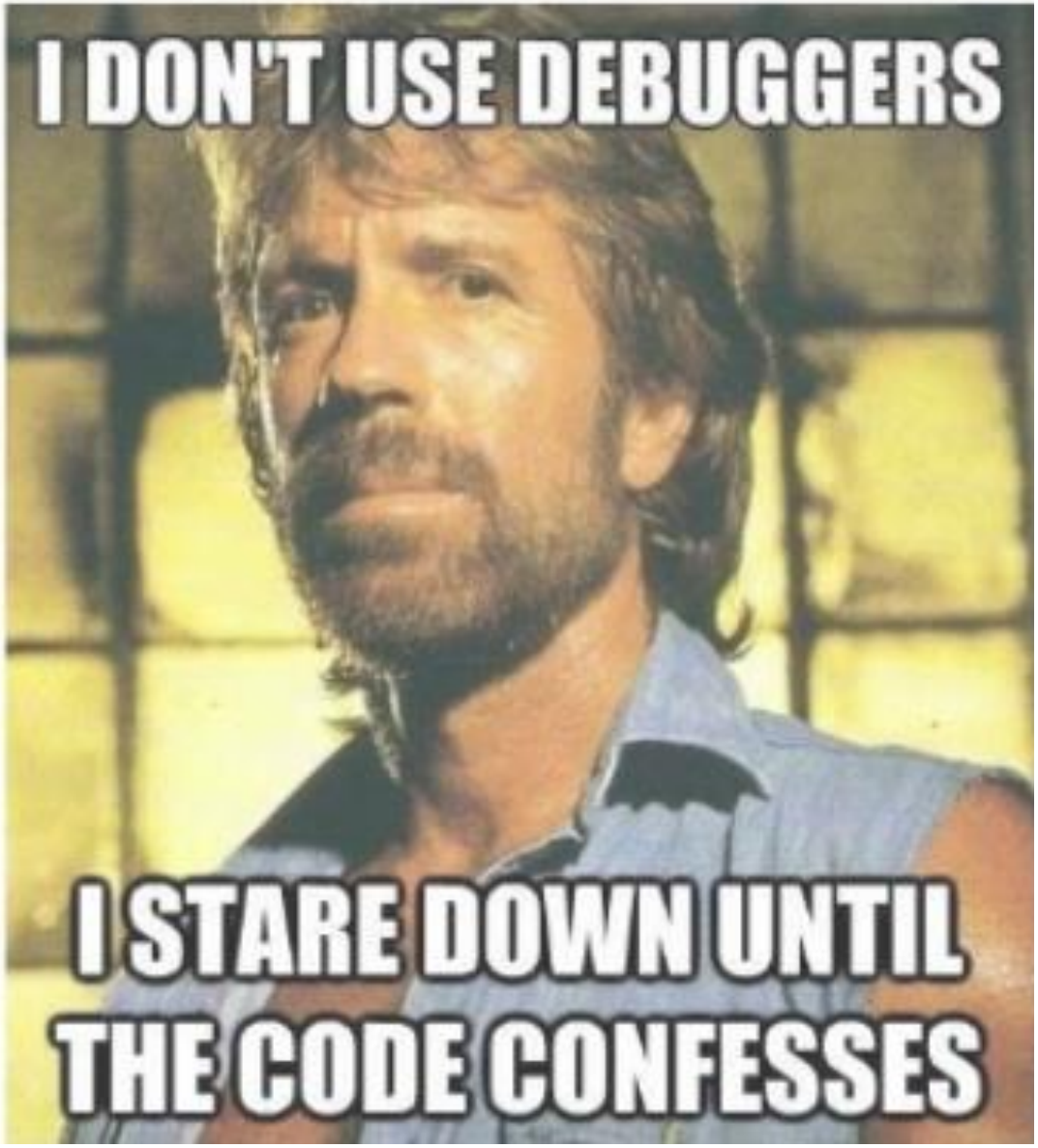
```
>>> joke_text = '%s: a device for finding furniture in the dark'
>>> bodypart1 = 'Knee'
>>> bodypart2 = 'Shin'
>>> print(joke_text % bodypart1)
Knee: a device for finding furniture in the dark
>>> print(joke_text % bodypart2)
Shin: a device for finding furniture in the dark
```

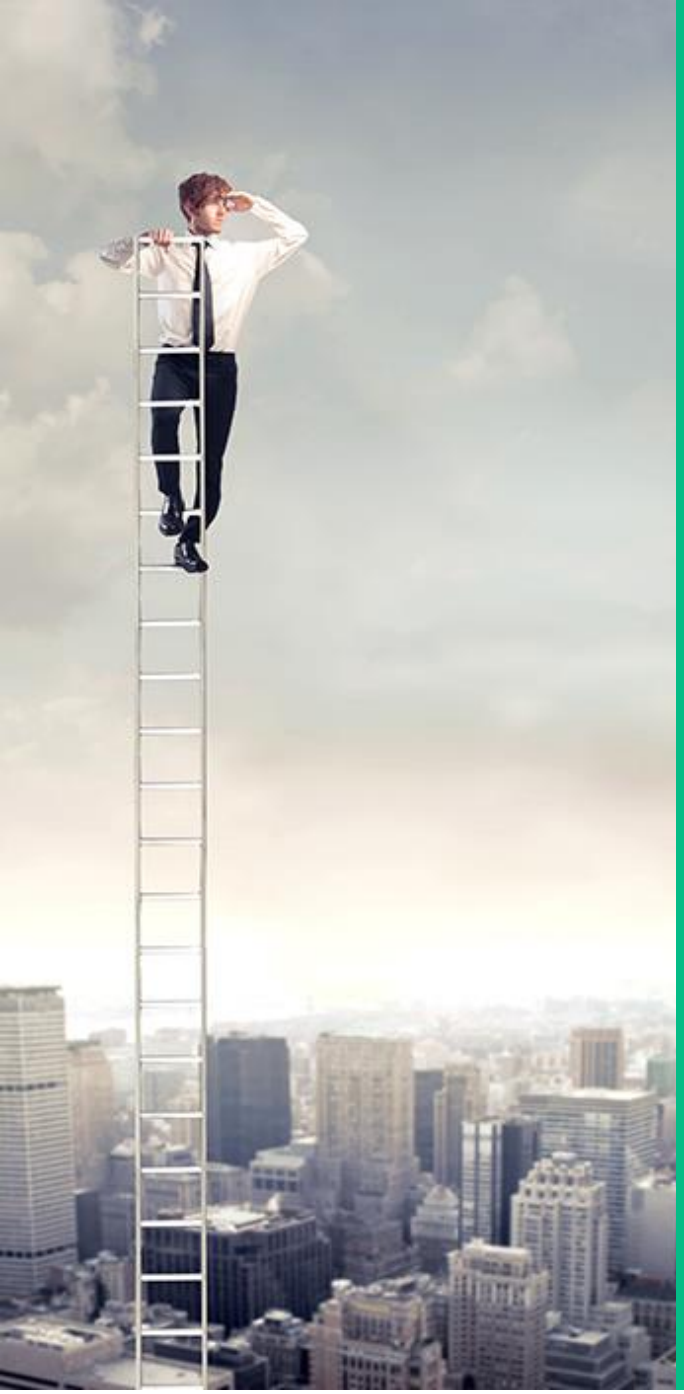
You can also use more than one placeholder in a string, like this:

```
>>> nums = 'What did the number %s say to the number %s? Nice belt!!'
>>> print(nums % (0, 8))
What did the number 0 say to the number 8? Nice belt!!
```

Source: J.R. Briggs, „Python for Kids”

DEBUGGER





Conditions

COMPARISON OPERATORS

The while loop executes as long as the condition (here: b < 10) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false.
The standard comparison operators are:

```
<   (less than),
>   (greater than),
==  (equal to),
<=  (less than or equal to),
>=  (greater than or equal to)
!=  (not equal to)
```


fun = 0
if booze:
fun +=
else:
fun -=

IF STATEMENTS

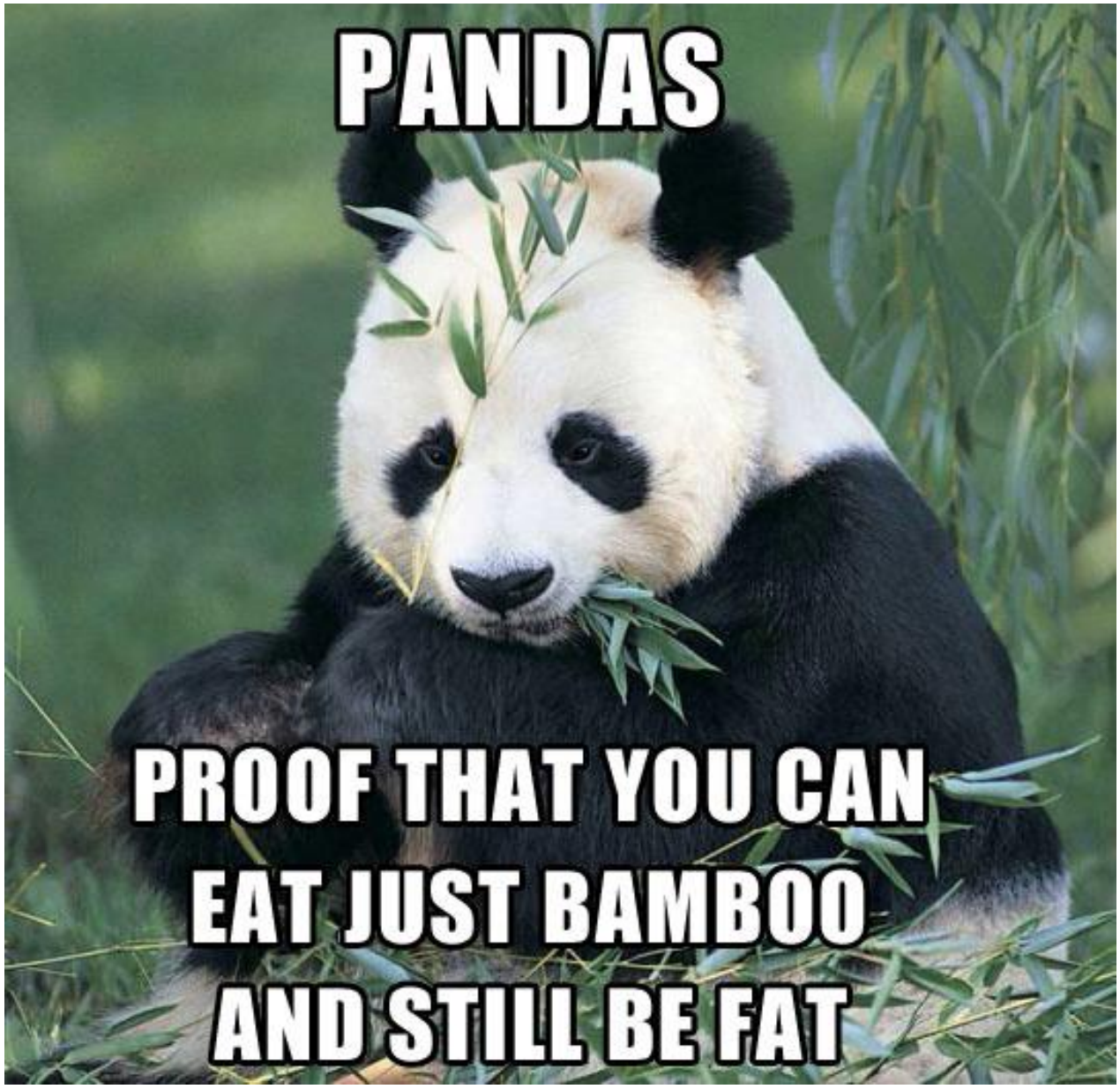
Perhaps the most well-known statement type is the **if statement**. For example:

```
x = int(input("Please enter an integer: "))
if x < 0:
     print('Negative changed to zero')
elif x == 0:
     print('Zero')
elif x == 1:
     print('Single')
else:
     print('More')
```

LOGICAL OPERATOR

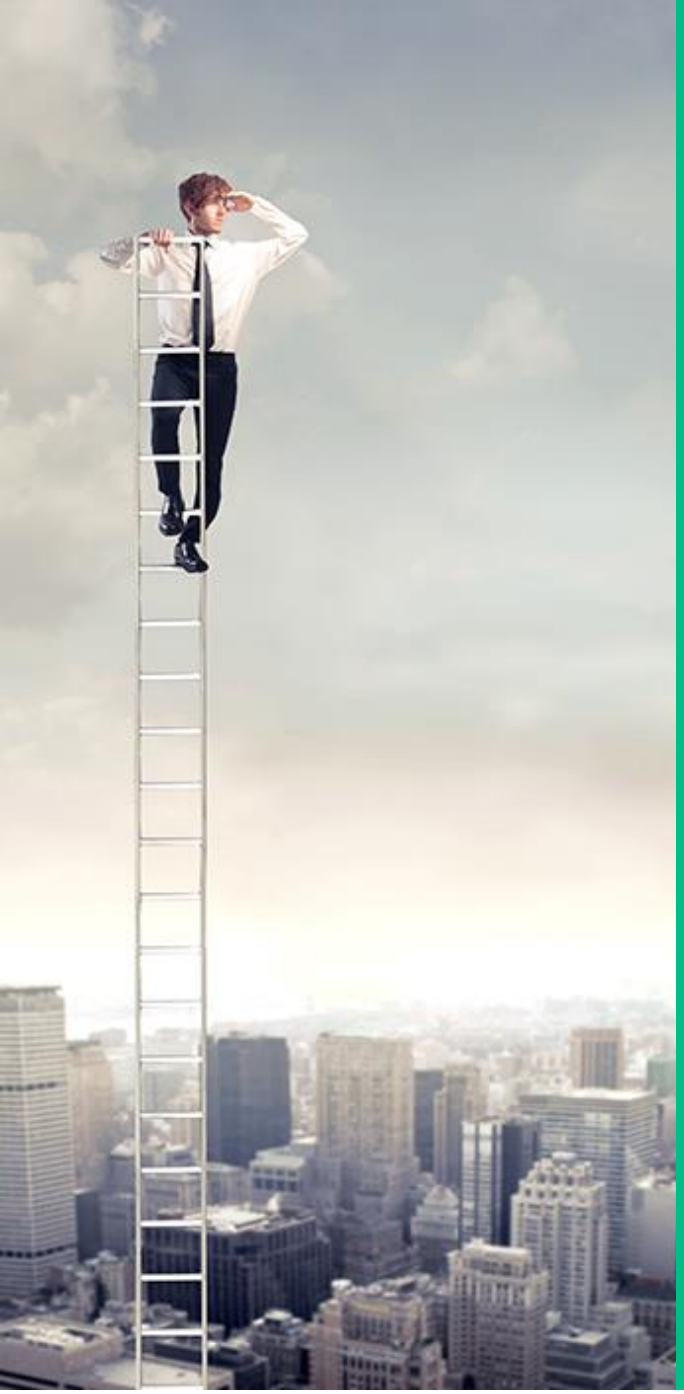
`and`	Determines whether both operands are true.	True and True is True True and False is False False and True is False False and False is False
`or`	Determines when one of two operands is true.	True or True is True True or False is True False or True is True False or False is False
`not`	Negates the truth value of a single operand. A true value becomes false and a false value becomes true.	not True is False not False is True

BMI – EXCERCIE



KOLEJNOŚĆ DZIAŁAŃ

()
**
~ + -
* / % //
+ -
>> <<
&
^ |
<= < > >=
== !=
= %= /= //= -= += *= **=
Is is not
In not in
not or and



Lists

LIST


PINK PANTHER'S
TO DO LIST:
-TO DO
-TO DO
-TO DO, TO DO, TO DO,
TO DO, TO DOOOO...
More pics on www.imfunny.net

LIST

Python knows a number of compound data types, used to group together other values. The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

```
squares = [1, 2, 4, 9, 16, 25]
```

Like strings (and all other built-in sequence type), lists can be indexed and sliced:

```
squares[0]   #indexing returns the item1
squares[-1] #returns last element
```

LIST

```
myList = [1, 'a', 3.14159, True]
```

myList

1	'a'	3.14159	True	
0	1	2	3	Index Forward
−4	−3	−2	−1	Index Backward

```
myList[1] → 'a'

myList[:3] → [1, 'a', 3.14159]
```

CONCATENATION

```
>>> squares = [1, 2, 4, 9, 16, 25]
>>> squares + [36, 49, 64, 81, 100]
```

CONCATENATION

Lists also supports operations like concatenation:

```
>>> squares = [1, 2, 4, 9, 16, 25]
>>> squares + [36, 49, 64, 81, 100]


[1, 2, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

CONCATENATION

Unlike strings, which are immutable, lists are a mutable type, i.e. it is possible to change their content:

```
cubes = [1, 8, 27, 65, 125]
cubes[3] = 64        # replace the value

cubes[1, 8, 27, 64, 125]
```

REPLACE

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
# replace some values
letters[2:5] = ['C', 'D', 'E']

letters['a', 'b', 'C', 'D', 'E', 'f', 'g']
```

REMOVE

 # now remove them

```
letters[2:5] = []

letters['a', 'b', 'f', 'g']
```

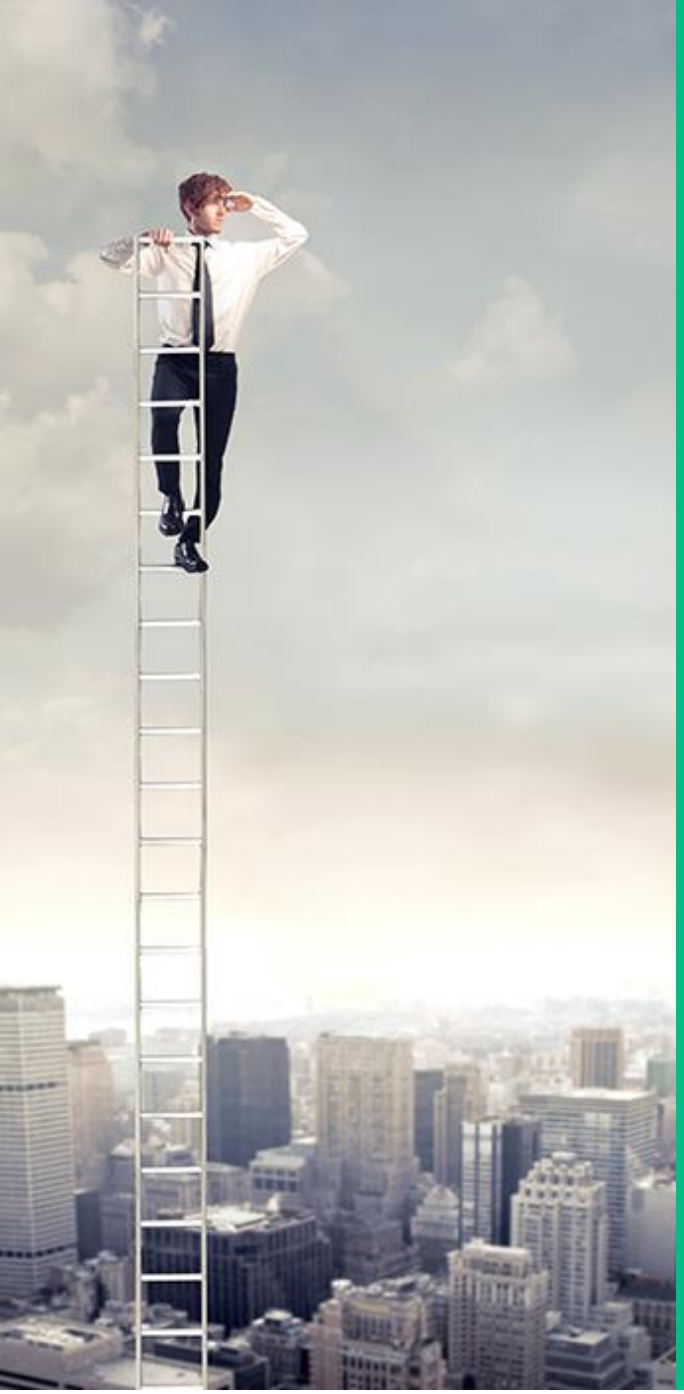
clear the list by replacing all the elements with an empty list

```
letters[:] = []
```

LEN()

The built-in function len() also applies to lists:

```
letters = ['a', 'b', 'c', 'd']

len(letters)

#is equal to 4!
```



Loops

FOR

Rather than always iterating over an arithmetic progression of numbers, or a the user the ability to define both the iteration step and halting condition, Python’s for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence.

The body of the loop is indented: indentation is Python's way of grouping statements.

FOR

#for statement example:

```
words = ['cat', 'dog', 'mouse']
for w in words:
        print(w, len(w))
```

WHILE

Fibonacci series example:
the sum of two elements defines the next...

```
a = 0
b = 1
while b < 10:
      print(b)
      temp = b
      b = a + b
      a = temp
```

RANGE FUNCTION

If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy.
It generates arithmetic progressions:

```
for i in range(5):
    print(i)
```

LIST EXAMPLE - APPEND

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> ????
```

LIST EXAMPLE - APPEND

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

RANGE

To iterate over the indices of a sequence, you can combine range() and len() as follows:

```
a = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(a)):
        print(i, a[i])
```

RANGE

The given end point is never part of the generated sequence; range(10) generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

```
range(5, 10)                    #5 through 9
range(0, 10, 3)                 #0, 3, 6, 9
range(-10, -100, -30)           #    ?
```

CONTINUE

```
iloscLiter = 0
slowo = "Geek Academy"

for i in slowo:
    if i == " ":
        print ("znalazlem spacje")
        continue
    iloscLiter += 1

print (iloscLiter)
```

BREAK

```
while True:
    var = input ("Enter something, or 'q' to quit: ")
    print (var)
    if var == 'q':
        break
```

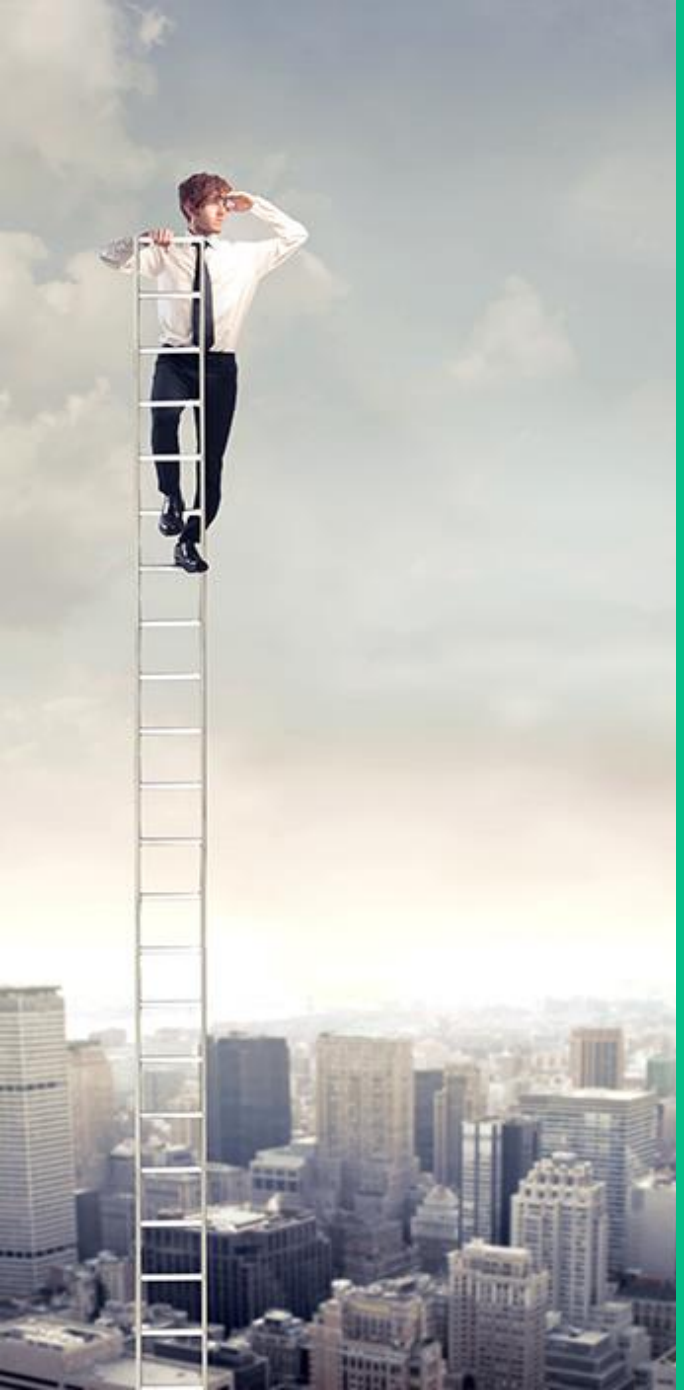
EXERCISE

read n

Paint the following shape (here n= 4):

```
a
aba
abcba
abcdcba
```

```
print("a",end="")
```



Tuple

TUPLE

Tuple are immutable, which means that you can not change it.

Creating tuple:

```
zwierzeta = ('pies', 'kot', 'mysz')

pusty = ()                    t1 = (10, )
```

TUPLE VS LIST

- Tuples are **faster** than lists. If you're defining a constant set of values and all you're ever going to do with it is iterate through it, use a tuple instead of a list.

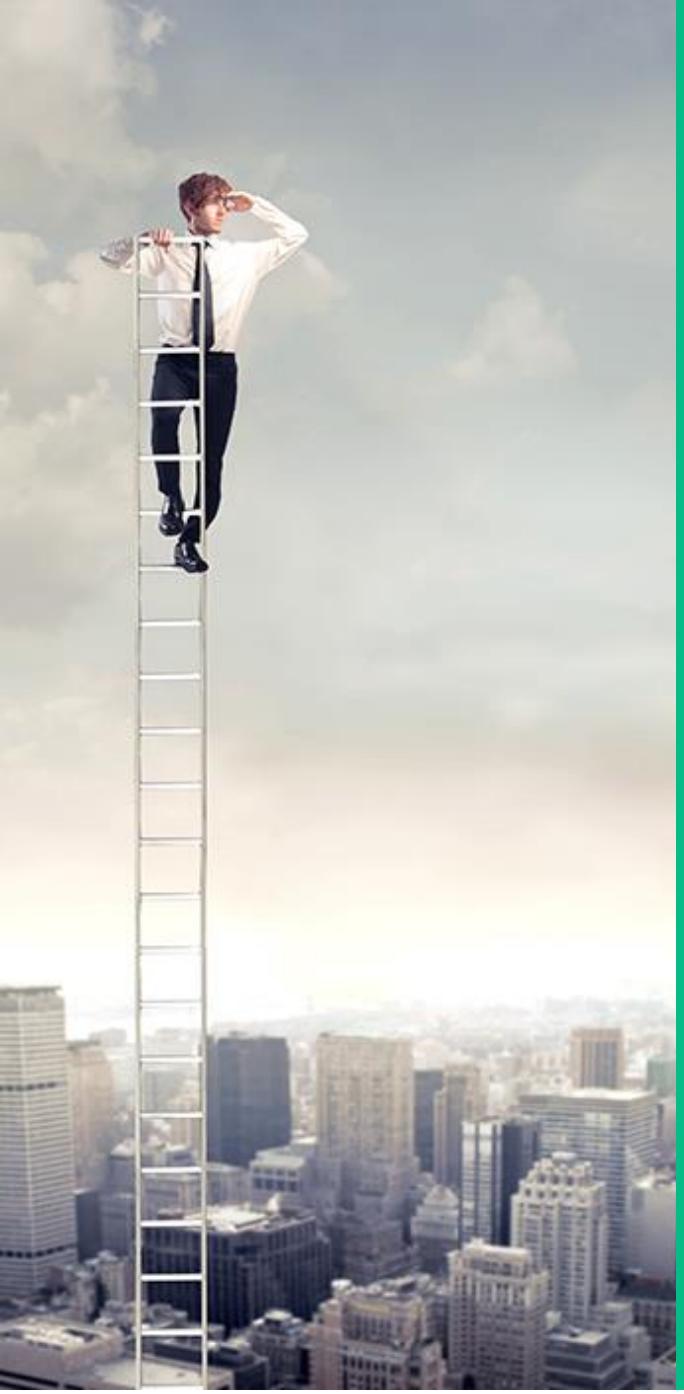
- It makes your code safer if you “**write-protect**” data that does not need to be changed. Using a tuple instead of a list is like having an implied assert statement that this data is constant, and that special thought (and a specific function) is required to override that.

- Some tuples can be used as **dictionary keys** (specifically, tuples that contain immutable values like strings, numbers, and other tuples). Lists can never be used as dictionary keys, because lists are not immutable.

TUPLE VS LIST

- Lists slower but more powerful than tuples
  - Lists can be modified, and they have lots of handy operations and mehtods
  - Tuples are immutable and have fewer features
- To convert between tuples and lists use the list() and tuple() functions:

```
li = list(tu)
tu = tuple(li)
```



Functions

FUNCTION

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

We can create a function:

```
def send_message():
    print ("that is my first function")
```

WRITE OWN FUNCTION

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
def fib(n):
    ….
```

PASSING ARGUMENTS

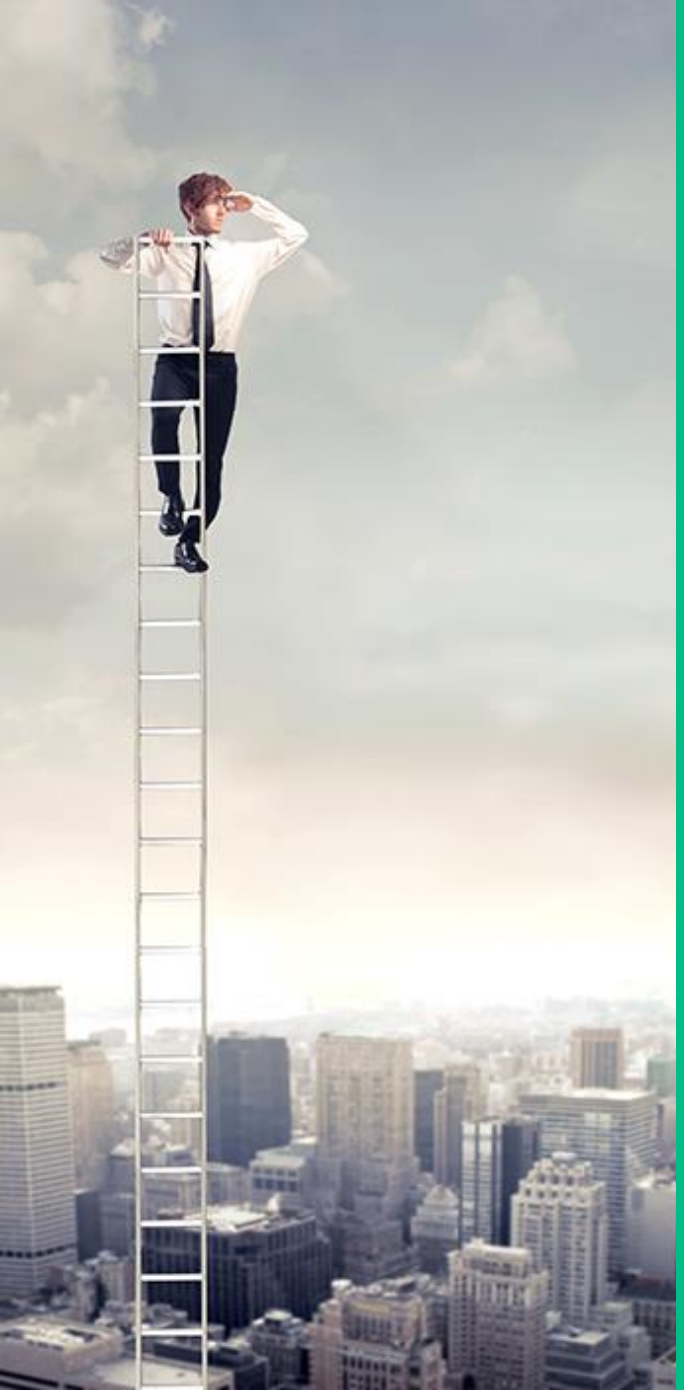
The most useful form is to specify a default value for one or more arguments.
This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
def ask_ok(prompt, retries=4, text='Yes'):
```

RETURN IN FUNCTION

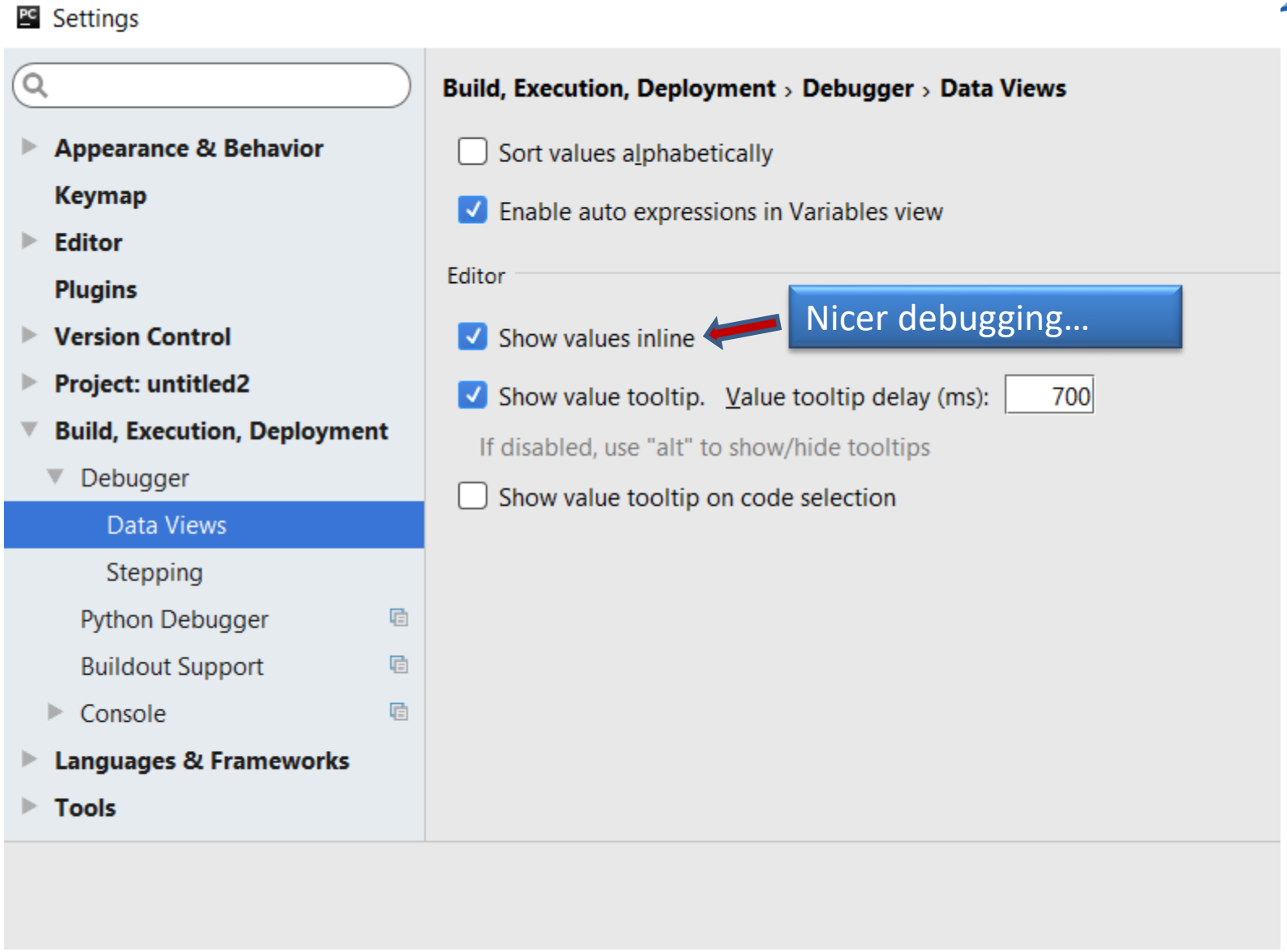
Function that return arguments:

```
def powerNumber(x):
    print("doing complicated calculation…")
    return x**2

print(powerNumber(16))
```



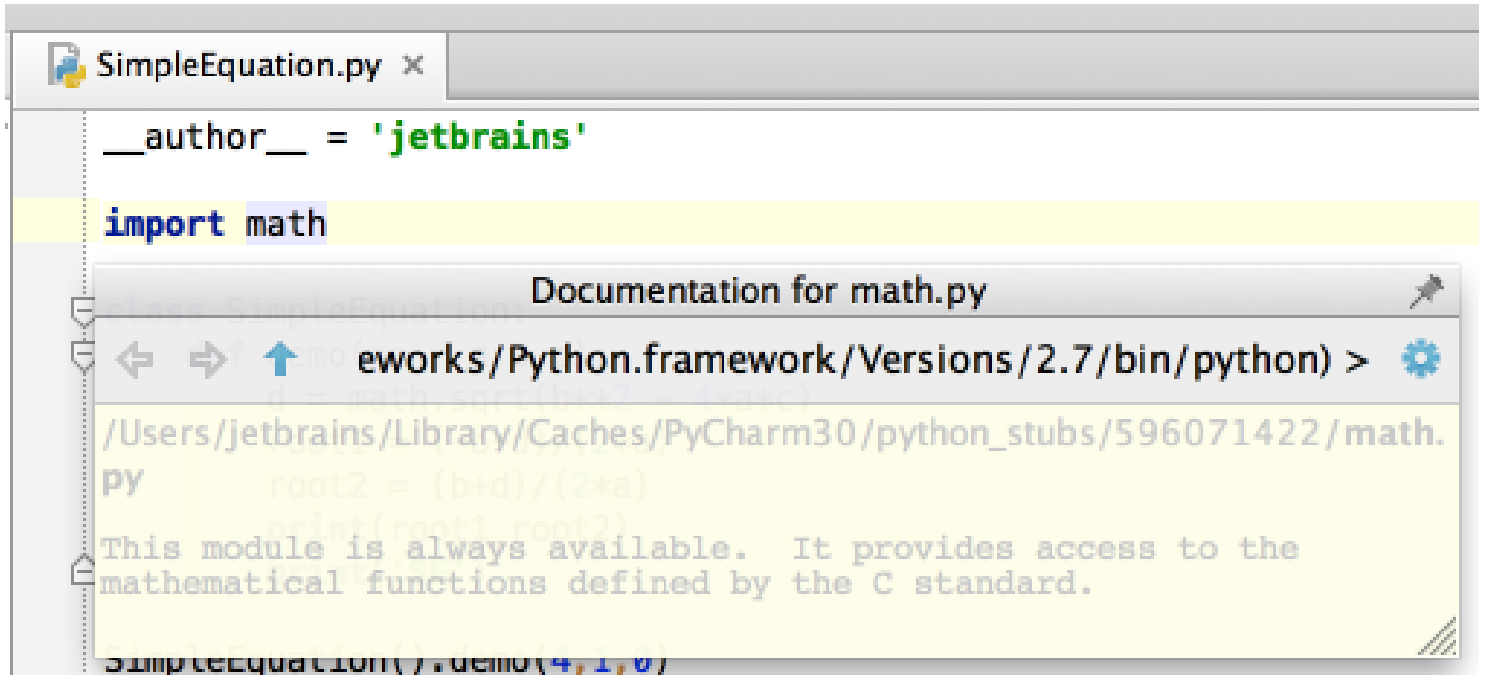
PyCharm Tips

PyCharm Tips



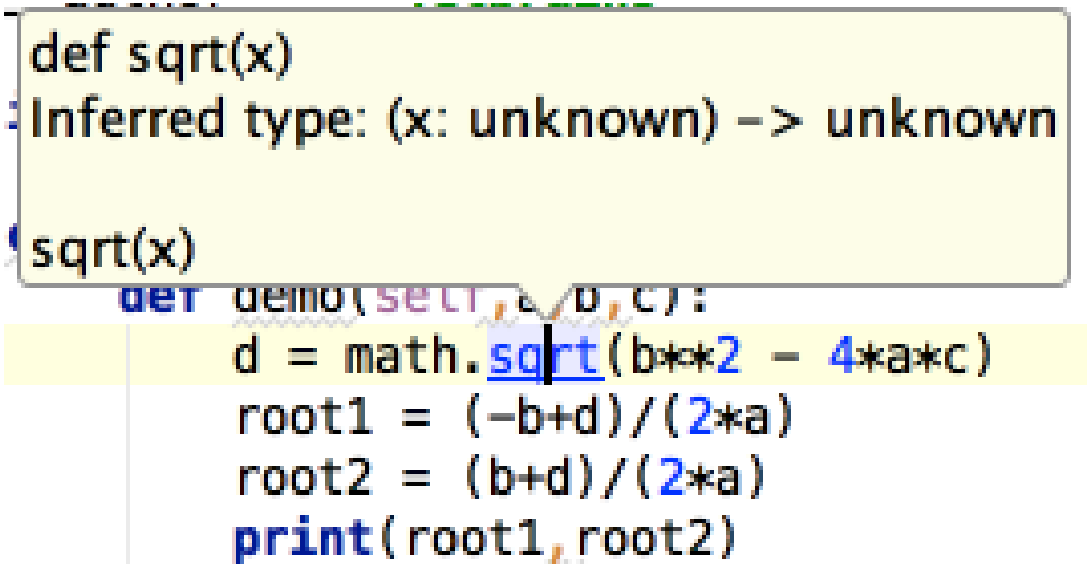
PyCharm Tips

To quickly see the **documentation** for the symbol at caret, press **Ctrl+Q** (View | Quick Documentation).



PyCharm Tips

To navigate to the declaration of a class, method or variable used somewhere in the code, position the caret at the usage and press **Ctrl+B.** You can also click the mouse on usages with the Ctrl key pressed to jump to declarations.



Ctrl+W (extend selection) in the editor selects the word at the caret and then selects expanding areas of the source code. For example, it may select a method name, then the expression that calls this method, then the whole statement, then the containing block, etc.

You can also select the word at the caret and the expanding areas of the source code by double-clicking the target areas in the editor.

PyCharm Tips

The **Code | Move Statement Up/Down** action is useful for reorganizing the code lines in your file, e.g., for bringing a variable declaration closer to the variable usage.
For example, select a code fragment and press **Ctrl+Shift+Up or Ctrl+Shift+Down.**

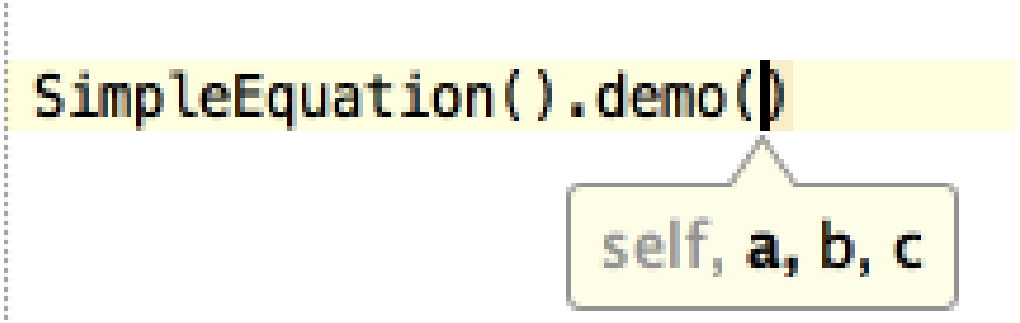
Before:

```
if text is  None:
    text = "None is"
    text += " just nothing"
else:
    text += "meaningful"
print(text)
```

After moving the lines up:

```
text = "None is"
text += " just nothing"
if text is  None:
    pass
else:
    text += "meaningful"
print(text)
```

If the cursor is between the parentheses of a method call, pressing **Ctrl+P** brings up a list of **valid parameters**.



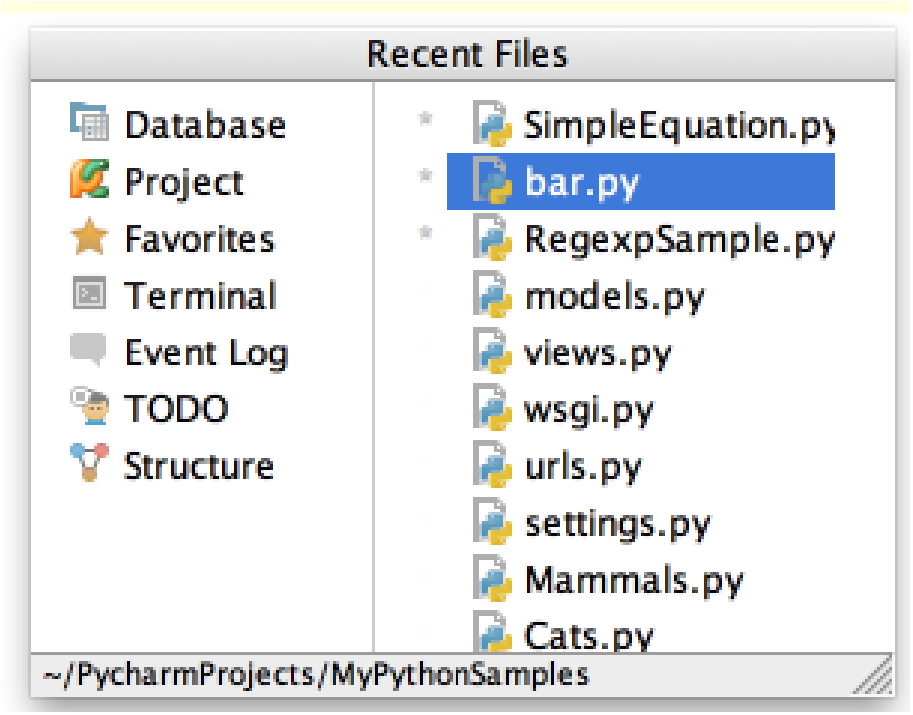
Ctrl+Shift+Backspace (Navigate | Last Edit Location) brings you back to the **last place** where you made changes in the code.

Pressing Ctrl+Shift+Backspace a few times moves you deeper into your **changes history**.

To see your local history of changes in a file, invoke Local **History | Show History** from the **context menu**. You can navigate through different file versions, see the differences and **roll back** to any previous version.

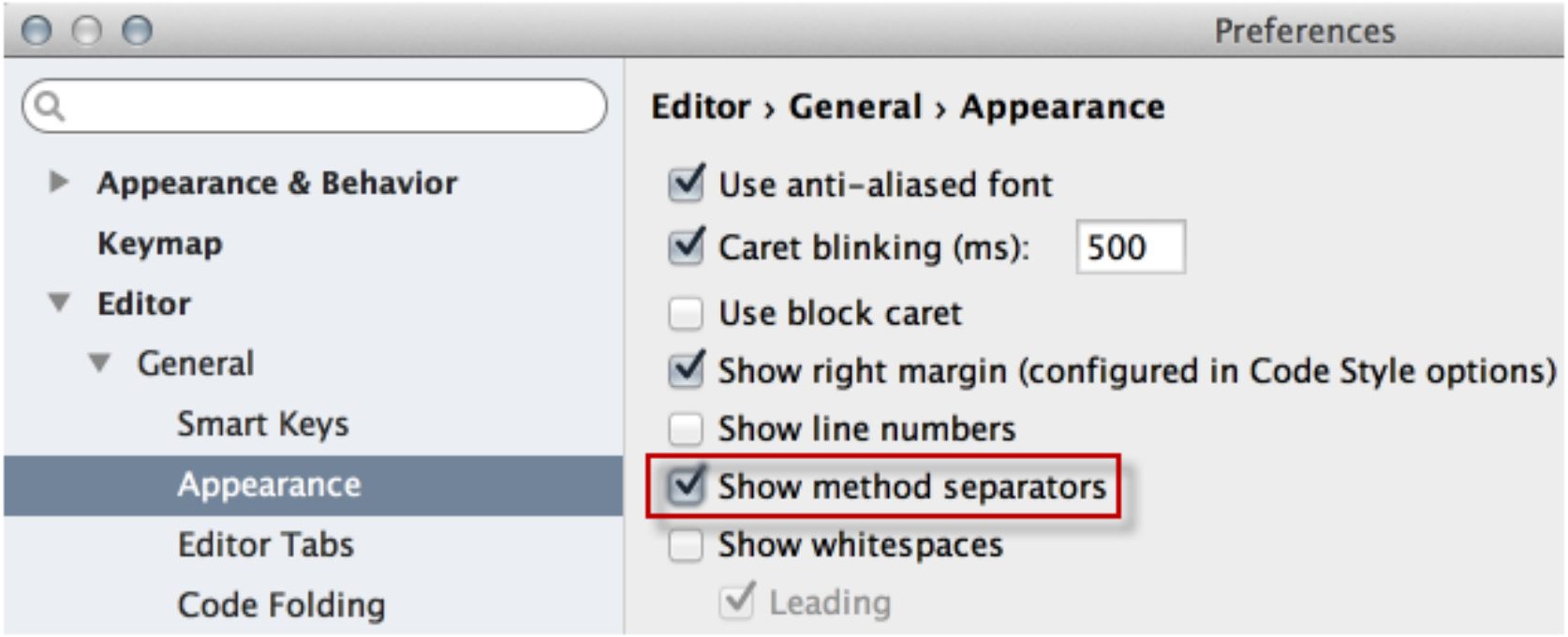
- Use the same context menu item to see the history of **changes** on a directory.

Ctrl+E (View | Recent Files) brings a popup list of the **recently visited files**. Choose the desired file and press Enter to open it.



PyCharm Tips

To show **separator lines between methods** in the editor, open the editor settings and select the Show method separators check box in the Appearance page.



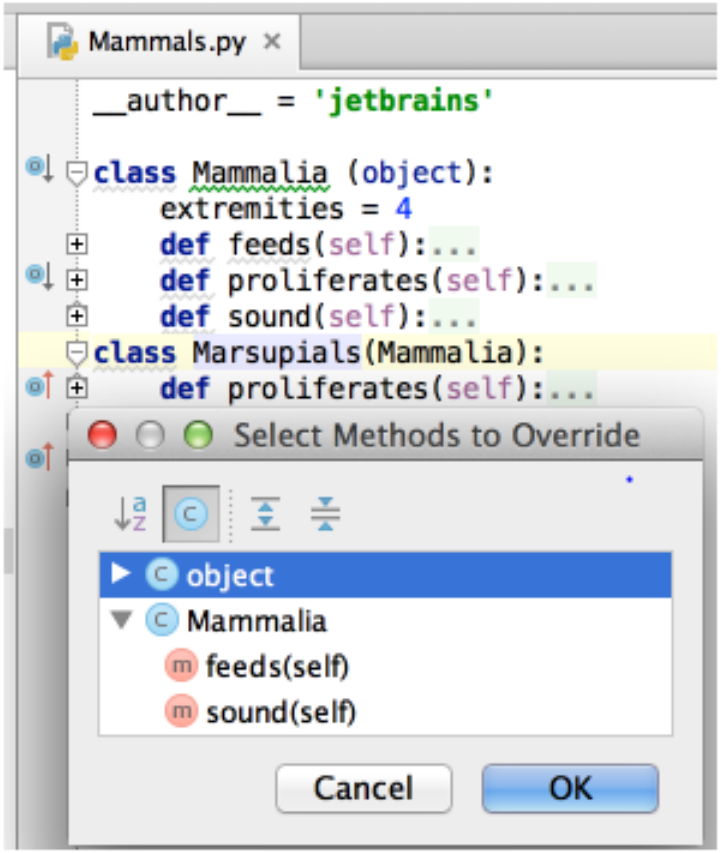
PyCharm Tips

- Use Alt+Up and Alt+Down keys to **quickly move between methods** in the editor.

- Use the Ctrl+Shift+V shortcut to choose and insert **recent clipboard** contents into the text.

PyCharm Tips

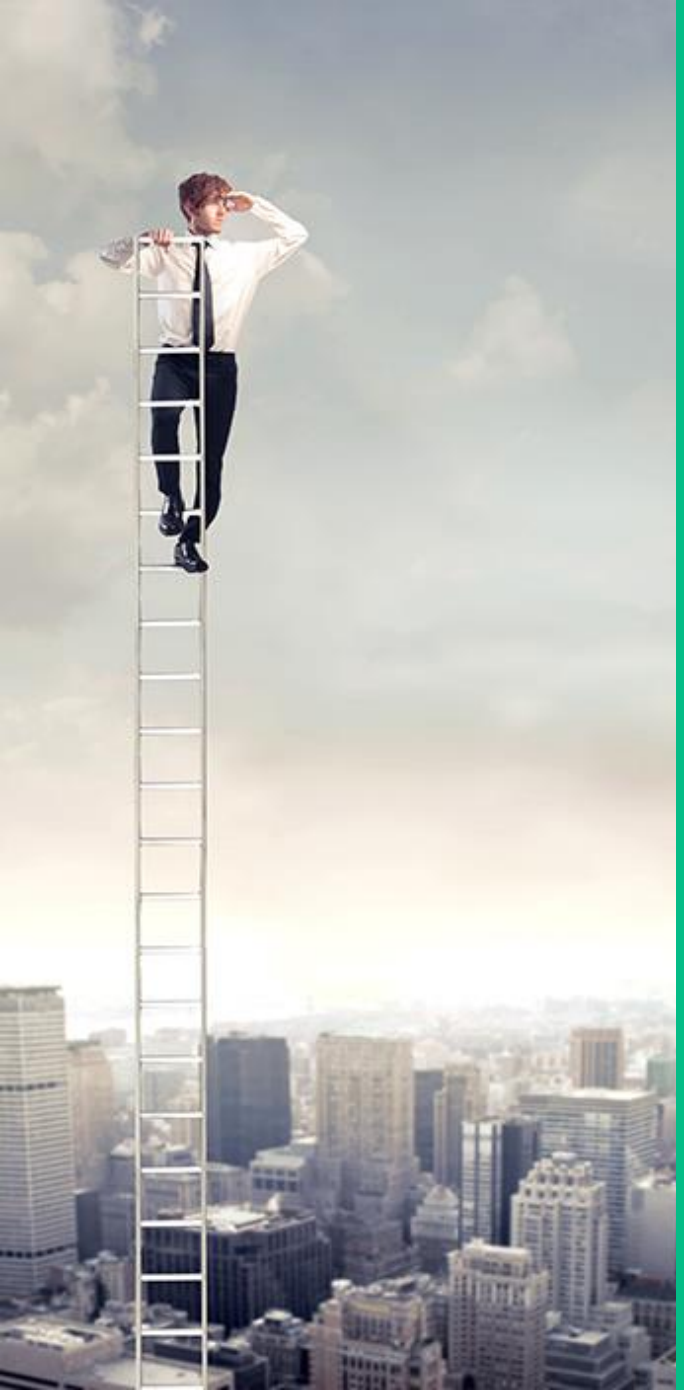
You can easily **override the methods** of the base class by pressing Ctrl+O (Code | Override Methods).



EXERCISE

read x,
read y,
paint the following shape (here exampe for x =6, y = 4):

```
   1  2  3  4  5  6
  *************
1 *  *  *  *  *  *  *
  *************
2 *  *  *  *  *  *  *
  *************
3 *  *  *  *  *  *  *
  *************
4 *  *  *  *  *  *  *
  *************
```



String Manipulation

A raw string completely ignores all escape characters and prints any backslash that appears in the string.

```
>>> print(r'That is Carol\'s cat.')
That is Carol\'s cat.
```

Multiline Strings

```
print('''Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,
Bob''')
```

```
Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,
Bob
```

An expression with two strings joined using in or not in will evaluate to a Boolean True or False.

```
>>> 'Hello' in 'Hello World'
True
>>> 'Hello' in 'Hello'
True
>>> 'HELLO' in 'Hello World'
False
>>> '' in 'spam'
True
>>> 'cats' not in 'cats and dogs'
False
```

STRING MANIPULATION

```
>>> spam = 'Hello world!'
>>> spam = spam.upper()
>>> spam
'HELLO WORLD!'
>>> spam = spam.lower()
>>> spam
'hello world!'
```

```
>>> spam = 'Hello world!'
>>> spam.islower()
False
>>> spam.isupper()
False
>>> 'HELLO'.isupper()
True
>>> 'abc12345'.islower()
True
>>> '12345'.islower()
False
>>> '12345'.isupper()
False
```

STRING MANIPULATION

```
>>> 'Hello world!'.startswith('Hello')
True
>>> 'Hello world!'.endswith('world!')
True
>>> 'abc123'.startswith('abcdef')
False
>>> 'abc123'.endswith('12')
False
>>> 'Hello world!'.startswith('Hello world!')
True
>>> 'Hello world!'.endswith('Hello world!')
True
```

STRING MANIPULATION

```
>>> ', '.join(['cats', 'rats', 'bats'])
'cats, rats, bats'
>>> ' '.join(['My', 'name', 'is', 'Simon'])
'My name is Simon'
>>> 'ABC'.join(['My', 'name', 'is', 'Simon'])
'MyABCnameABCisABCSimon'
```

STRING MANIPULATION

```
>>> 'My name is Simon'.split()
['My', 'name', 'is', 'Simon']
```

```
>>> 'MyABCnameABCisABCSimon'.split('ABC')
['My', 'name', 'is', 'Simon']
>>> 'My name is Simon'.split('m')
['My na', 'e is Si', 'on']
```

STRING MANIPULATION

```
>>> 'Hello'.rjust(10)
'     Hello'
>>> 'Hello'.rjust(20)
'               Hello'
>>> 'Hello World'.rjust(20)
'         Hello World'
>>> 'Hello'.ljust(10)
'Hello     '
```

```
>>> 'Hello'.center(20)
'       Hello        '
>>> 'Hello'.center(20, '=')
'=======Hello========'
```

```
>>> 'Hello'.rjust(20, '*')
'***************Hello'
>>> 'Hello'.ljust(20, '-')
'Hello---------------'
```

STRING MANIPULATION

```
>>> help(str)
```

STRING MANIPULATION

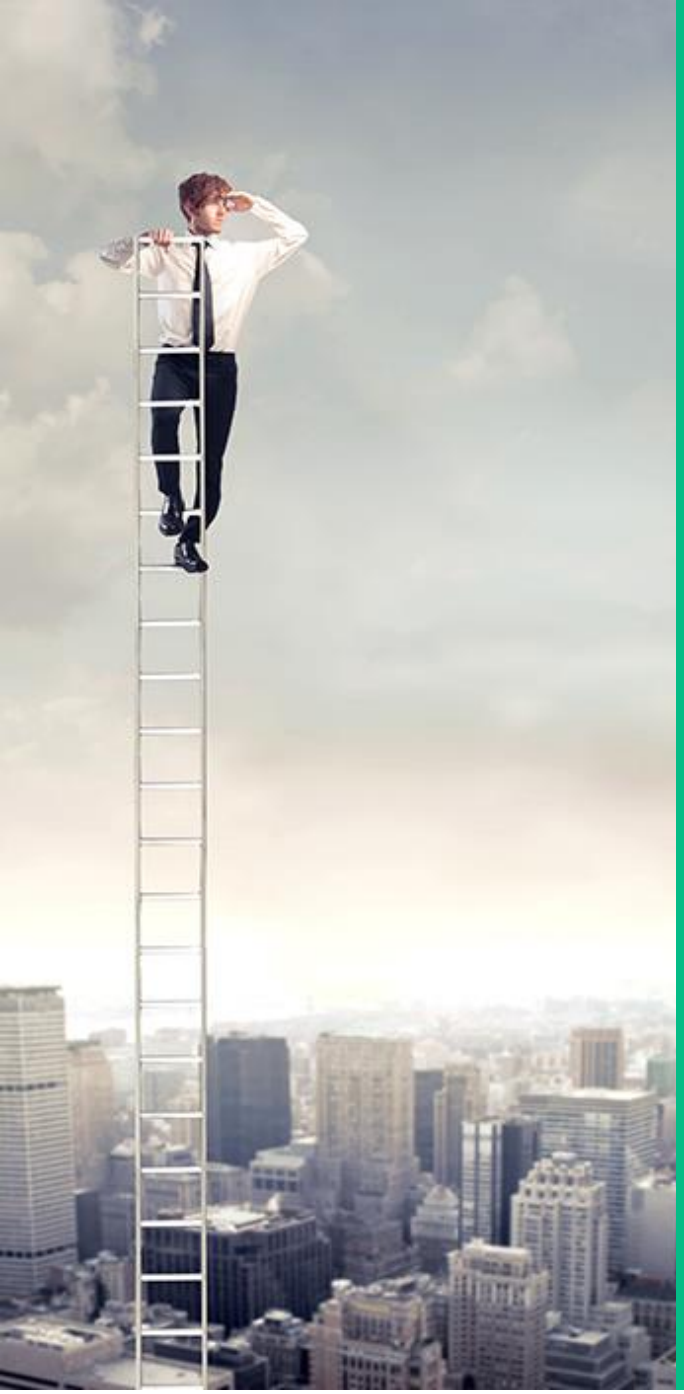
- **isalpha()** returns True if the string consists only of letters and is not blank.
- **isalnum()** returns True if the string consists only of letters and numbers and is not blank.
- **isdecimal()** returns True if the string consists only of numeric characters and is not blank.
- **isspace()** returns True if the string consists only of spaces, tabs, and newlines and is not blank.
- **istitle()** returns True if the string consists only of words that begin with an uppercase letter followed by only lowercase letters.

STRING MANIPULATION

- **isalpha()** returns True if the string consists only of letters and is not blank.
- **isalnum()** returns True if the string consists only of letters and numbers an
- **isdecimal()** returns True
numeric characters and
- **isspace()** returns True if
spaces, tabs, and newli
- **istitle()** returns True if t
words that begin with a
by only lowercase letters.

Task: read and validate input

- **Name**
- **Age**
- **Password**
- **Zip code**
- **Email**



List Manipulation

LIST METHODS

Lists have many methods, including index, count, remove, reverse, sort

```
>>> li = ['a', 'b', 'c', 'b']
>>> li.index('b')  # index of 1st occurrence
1
>>> li.count('b')  # number of occurrences
2
>>> li.remove('b') # remove 1st occurrence
>>> li
  ['a', 'c', 'b']
```

LIST METHODS

```
>>> li = [5, 2, 6, 8]

>>> li.reverse()     # reverse the list *in place*
>>> li
  [8, 6, 2, 5]

>>> li.sort()        # sort the list *in place*
>>> li
  [2, 5, 6, 8]

>>> li.sort(some_function)
    # sort in place using user-defined comparison
```

LIST METHODS

- *+* creates a fresh list with a new memory ref
- *extend* operates on list `li` in place.

```
>>> li.extend([9, 8, 7])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

- *Potentially confusing*:
  - *extend* takes a list as an argument.
  - *append* takes a singleton as an argument.

```
>>> li.append([10, 11, 12])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [10,
   11, 12]]
```

LIST OF LISTS

```
myLst = ['a', [1, 2, 3], 'z']
```

- What is the second element (index 1) of that list?
- Another list:

```
myLst[1][0] #apply left to right
myLst[1] => [1, 2, 3]
[1, 2, 3][0] => 1

[1, [2,[3,4]], 5][1][1][0]=>?
```

"*" OPERATOR

- The * operator produces a *new* tuple, list, or string that “repeats” the original content.

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)

>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> “Hello” * 3
‘HelloHelloHello’
```

IN - EXAMPLE

- Boolean test whether a value is inside a container:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

- For strings, tests for substrings

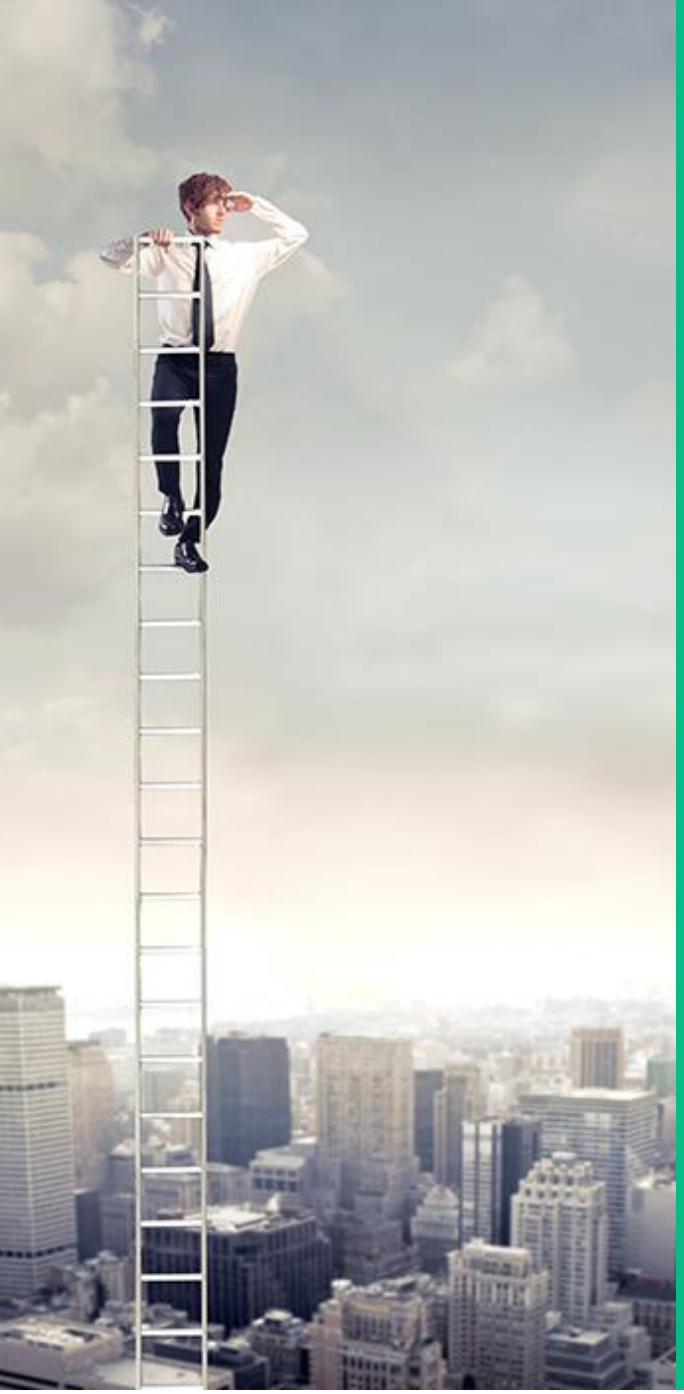
```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

- Be careful: the *in* keyword is also used in the syntax of *for loops* and *list comprehensions*

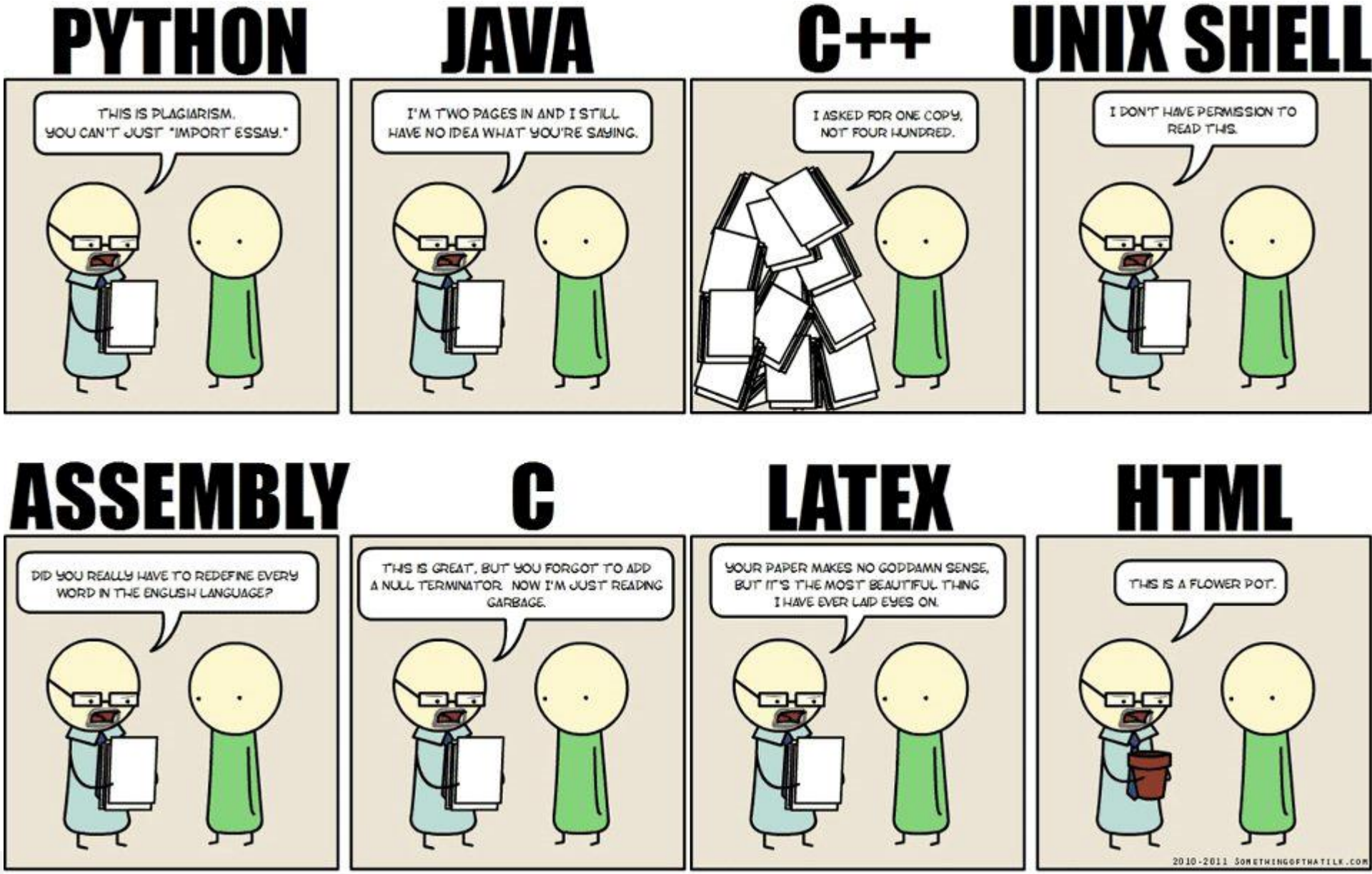
LIST METHODS

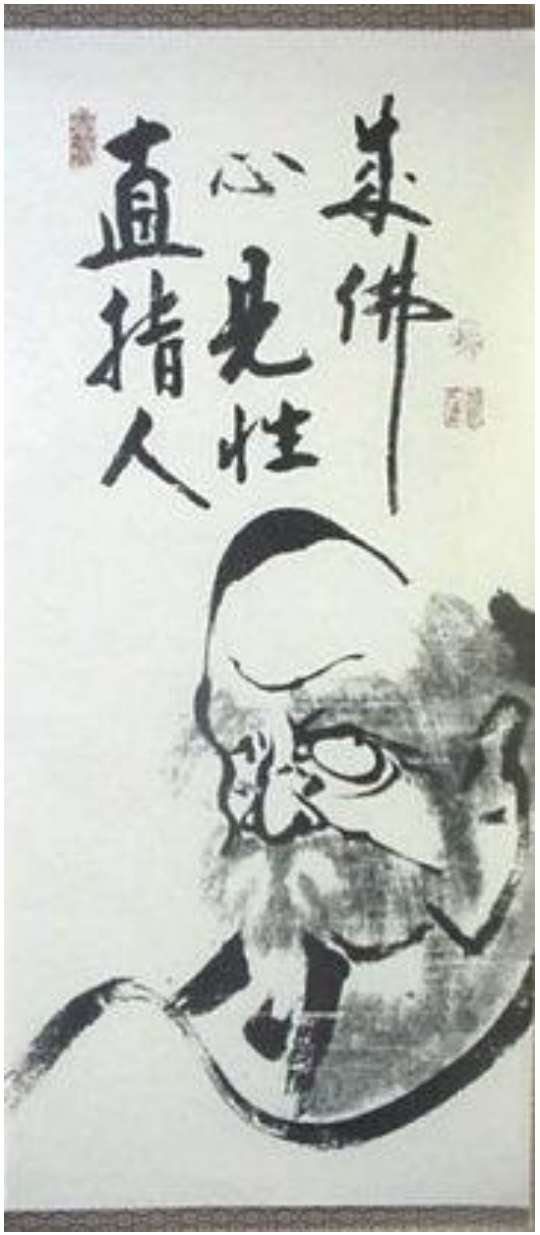
Here are all of the methods of list objects:

list.append(x) Add an item to the end of the list.
list.extend(L) Extend the list by appending all the items in the given list.
list.insert(i, x) Insert an item at a given position.
list.remove(x) Remove the first item from the list whose value is x.
list.pop([i]) Remove the item at the given position in the list, and return it.
list.index(x) Return the index in the list of the first item whose value is x**.**
list.count(x) Return the number of times x appears in the list.
list.sort() Sort the items of the list in place.
list.reverse() Reverse the elements of the list in place.



Modules


PYTHON
THIS IS PLAGIARISM. YOU CAN'T JUST "IMPORT ESSAY."
JAVA
I'M TWO PAGES IN AND I STILL HAVE NO IDEA WHAT YOU'RE SAYING.
C++
I ASKED FOR ONE COPY, NOT FOUR HUNDRED.
UNIX SHELL
I DON'T HAVE PERMISSION TO READ THIS.
ASSEMBLY
DID YOU REALLY HAVE TO REDEFINE EVERY WORD IN THE ENGLISH LANGUAGE?
C
THIS IS GREAT, BUT YOU FORGOT TO ADD A NULL TERMINATOR. NOW I'M JUST READING GARBAGE.
LATEX
YOUR PAPER MAKES NO GODDAMN SENSE, BUT IT'S THE MOST BEAUTIFUL THING I HAVE EVER LAID EYES ON.
HTML
THIS IS A FLOWER POT.
2010-2011 SOMETHINGOFTHATILK.COM



Zen (jap. 禅, cicha medytacja, religijna kontemplacja) – nurt buddyzmu, który w pełni rozwiniętą formę uzyskał w Chinach, skąd przedostał się do Korei, Wietnamu i Japonii.

Przez zen rozumie się także "spokojny umysł", "kontrolowanie procesu myślenia", "przyswajanie koncepcji". Jest to koncentracja myśli na jednym punkcie i kontemplowanie prawdy życia poprzez praktykę siedzącej medytacji. Dzięki praktyce zen umysł odzyskuje świeżość, a myślenie staje się przejrzyste niczym woda. Innymi słowy, zen to wyzwolona świadomość, narzędzie i ostateczne schronienie.

Zen of Python
import this

LIST OF BUILD IN PYTHON MODULES

List of all build-in modules:
https://docs.python.org/3/py-modindex.html

Python Module Index

_ | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | z

_	
__future__	*Future statement definitions*
__main__	*The environment where the top-level script is run.*
_dummy_thread	*Drop-in replacement for the _thread module.*
_thread	*Low-level threading API.*
a	
abc	*Abstract base classes according to PEP 3119.*
aifc	*Read and write audio files in AIFF or AIFC format.*
argparse	*Command-line option and argument parsing library.*
array	*Space efficient arrays of uniformly typed numeric values.*
ast	*Abstract Syntax Tree classes and manipulation.*
asynchat	*Support for asynchronous command/response protocols.*
asyncio	*Asynchronous I/O, event loop, coroutines and tasks.*
asyncore	*A base class for developing asynchronous socket handling services.*
atexit	*Register and execute cleanup functions.*
audioop	*Manipulate raw audio data.*

DATATIME

```
import datetime

print(datetime.datetime.now())
print(datetime.datetime.now().hour)
print(datetime.datetime.now().day)
print(datetime.datetime.now().year)
print(datetime.datetime.now().month)
```

IMPORT MODULE

Example of importing external module:

```
import math

#calucalting sin of 1 radian
x = math.sin(1)
print(x)
```

IMPORT MODULE

```
import database
db = database.Database()
# Do queries on db
```

```
from database import Database
db = Database()
# Do queries on db
```

```
from database import Database as DB
db = DB()
# Do queries on db
```

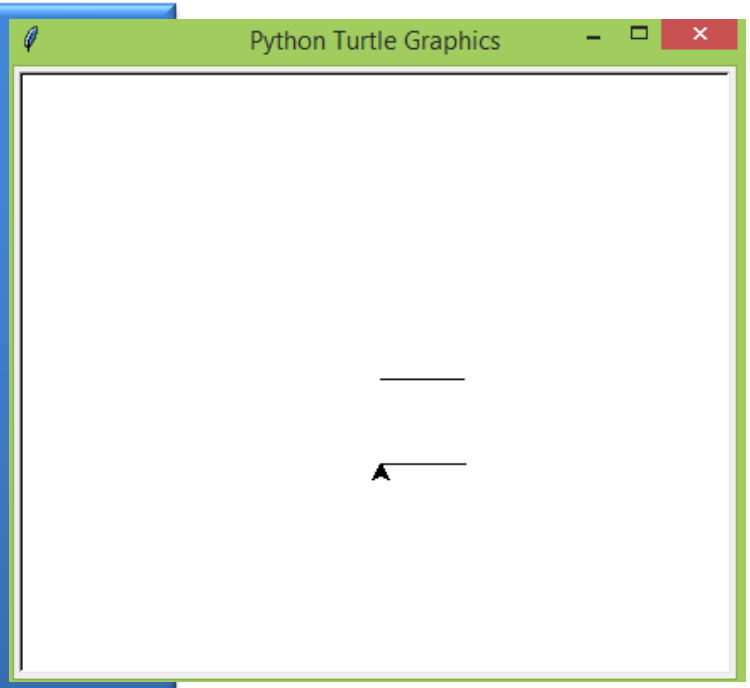
```
from database import *
```

EXERCICES FOR FUN! ☺

Example of using turtle:

```
import turtle
t = turtle.Pen()
t.forward (50)
t.right(90)
t.up()
t.forward (50)
t.right(90)
t.down()
t.forward (50)
t.right(90)
turtle.Screen().exitonclick()
```

Program results:

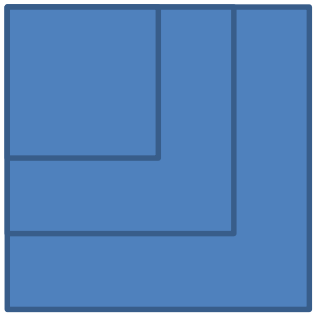


EXERCICES FOR FUN! ☺

Using turtle write function that draws following shapes:

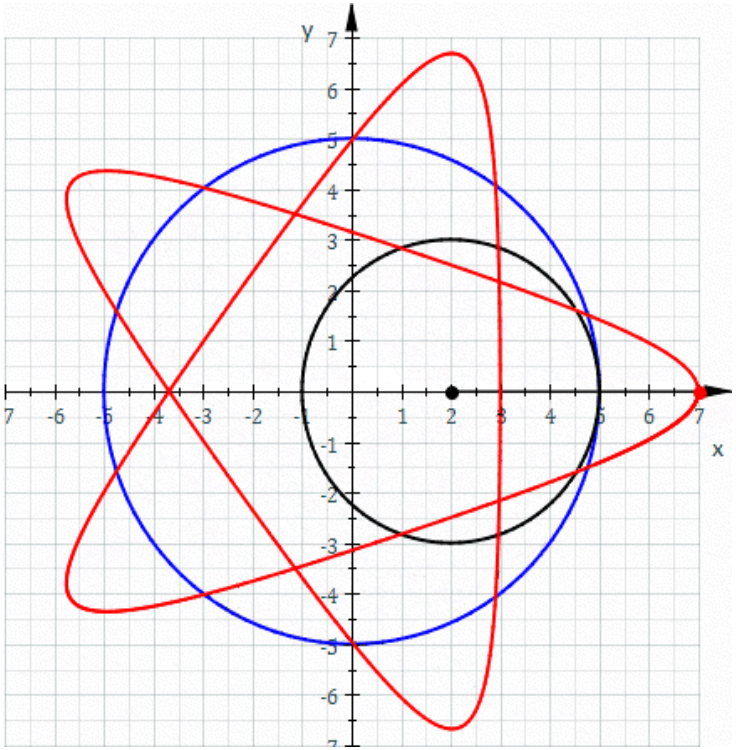
Triangle (size)

Rectangle (size)



EXERCICES FOR FUN! ☺

Using Turtle write function that draws Hypotrochoid (the following shape):



$$x = (R - r)\cos(t) + h\cos(\frac{R - r}{r}t)$$

$$y = (R - r)\sin(t) - h\sin(\frac{R - r}{r}t)$$

IMPORT RANDOM

random — Generate pseudo-random numbers
Source code: Lib/random.py
This module implements pseudo-random number generators for various distributions.

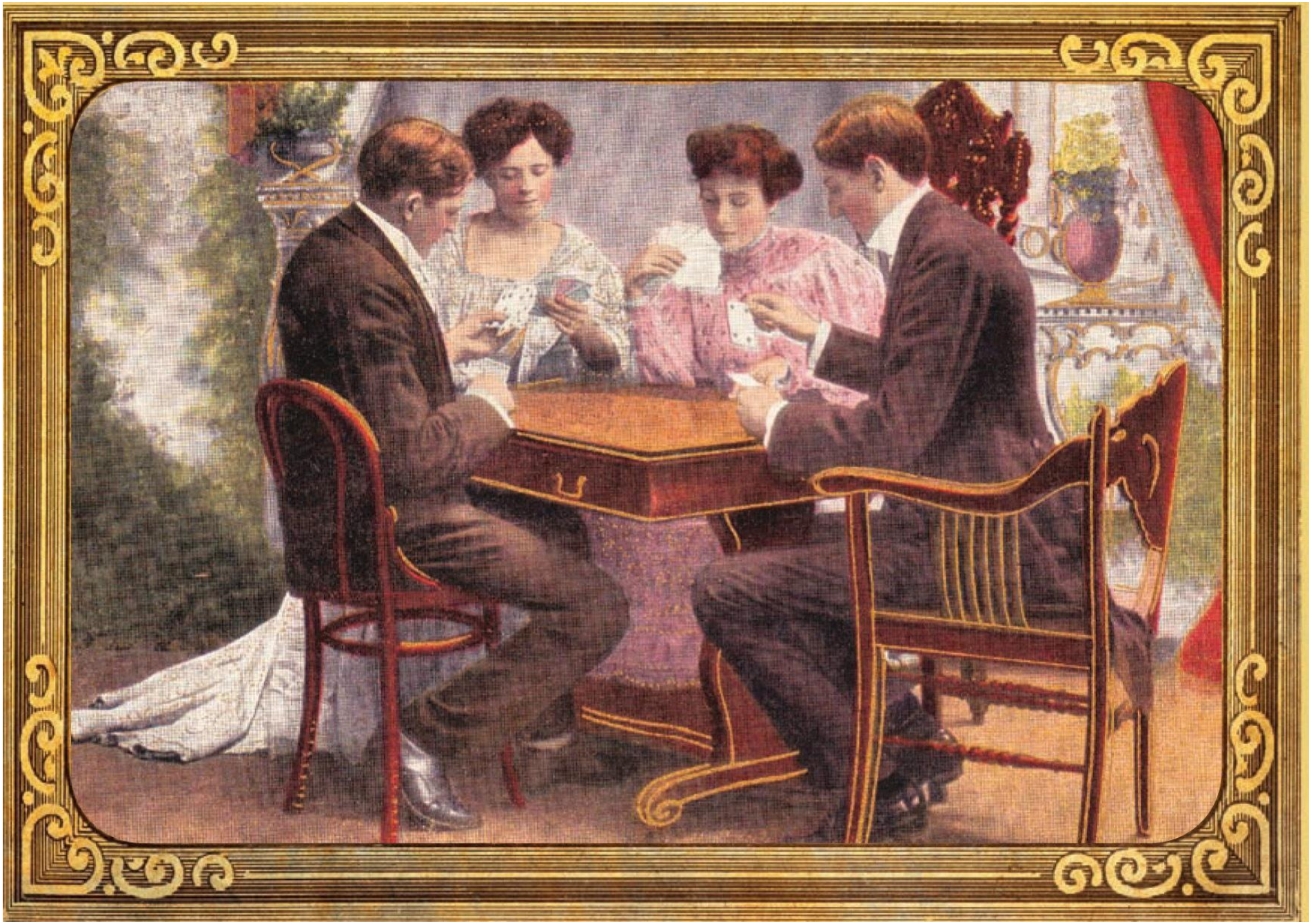
```
import random
```

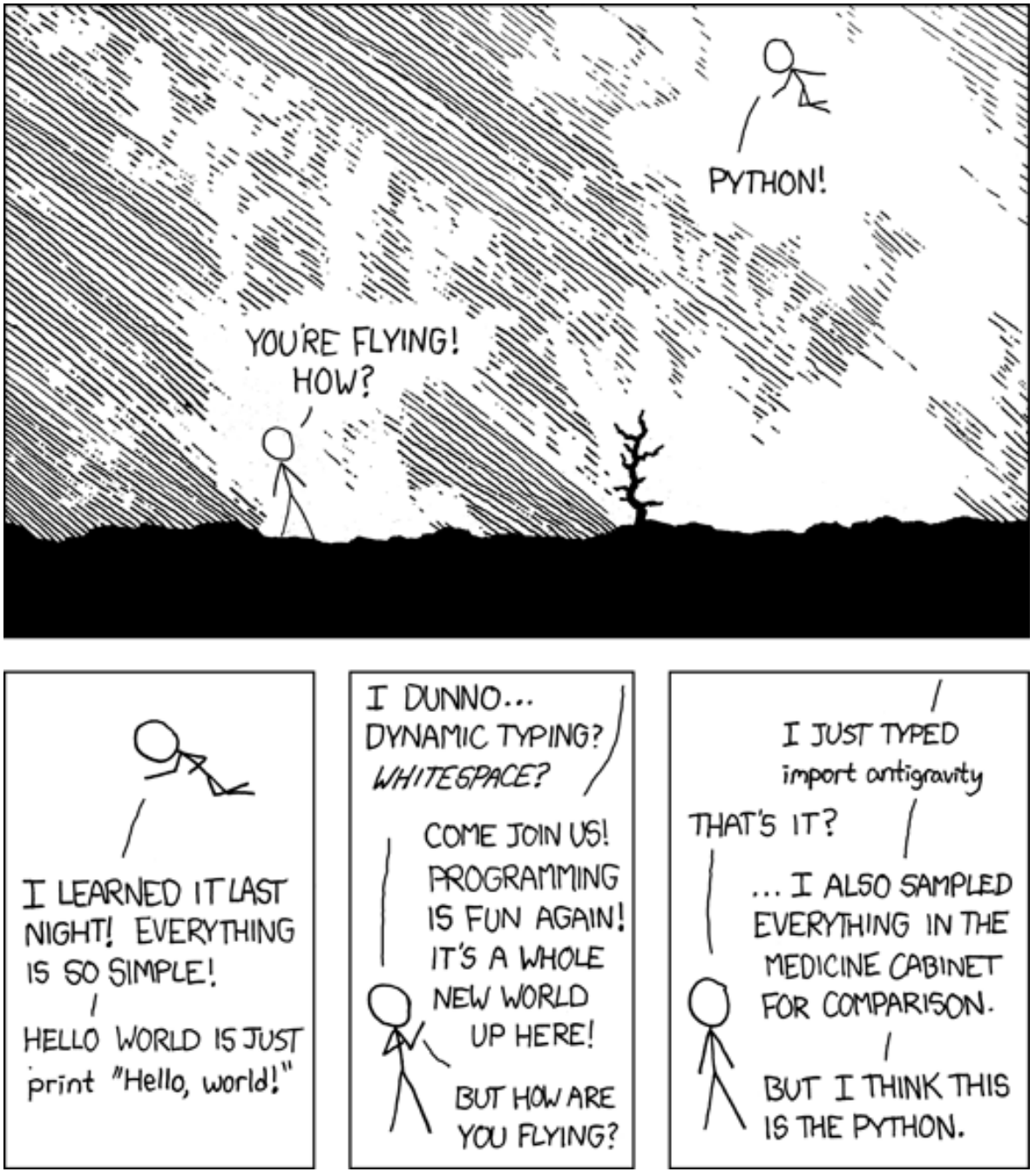
Warning: The pseudo-random generators of this module should not be used for security purposes. Use `os.urandom()` or `SystemRandom` if you require a cryptographically secure pseudo-random number generator.

Source: https://docs.python.org/2/library/random.html

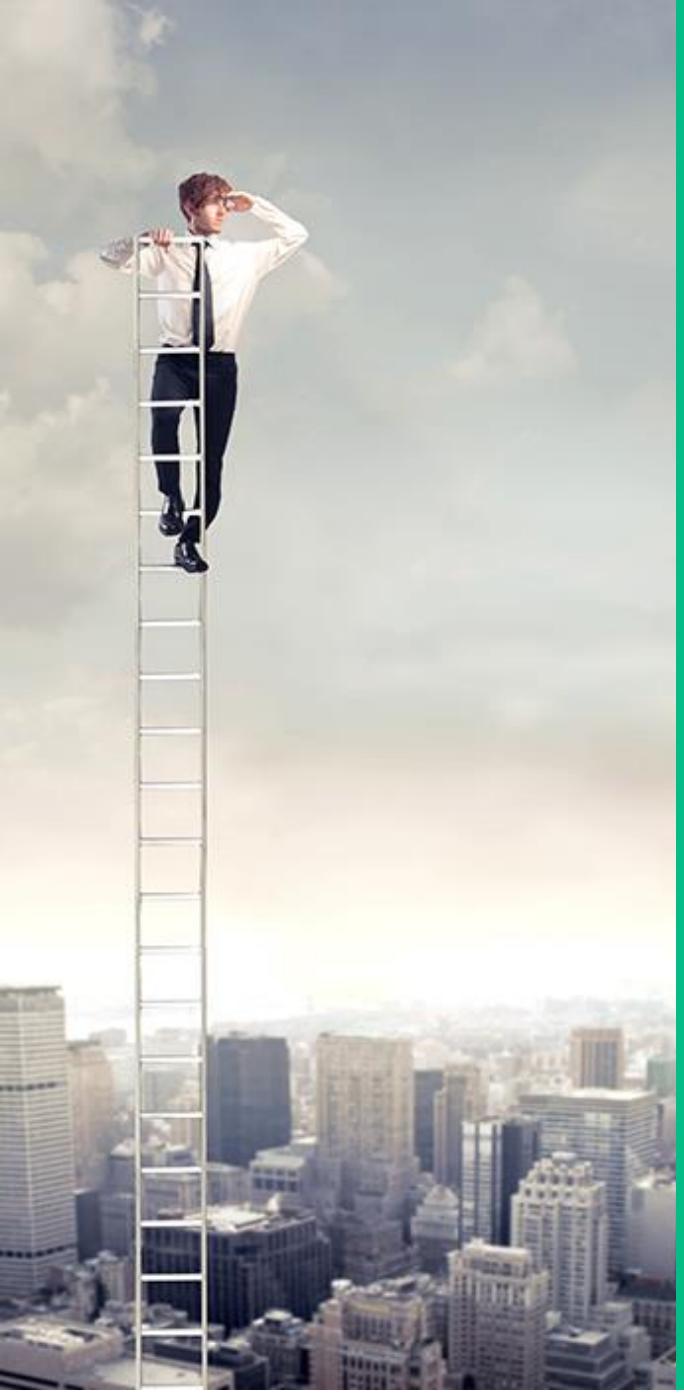
IMPORT RANDOM

```
>>> random.random()          # Random float x, 0.0 <= x < 1.0
0.37444887175646646
>>> random.uniform(1, 10)   # Random float x, 1.0 <= x < 10.0
1.1800146073117523
>>> random.randint(1, 10)   # Integer from 1 to 10, endpoints included
7
>>> random.randrange(0, 101, 2)   # Even integer from 0 to 100
26
>>> random.choice('abcdefghij')   # Choose a random element
'c'
>>> items = [1, 2, 3, 4, 5, 6, 7]
>>> random.shuffle(items)
>>> items
[7, 3, 2, 5, 6, 4, 1]
>>> random.sample([1, 2, 3, 4, 5],  3)   # Choose 3 elements
[4, 1, 5]
```





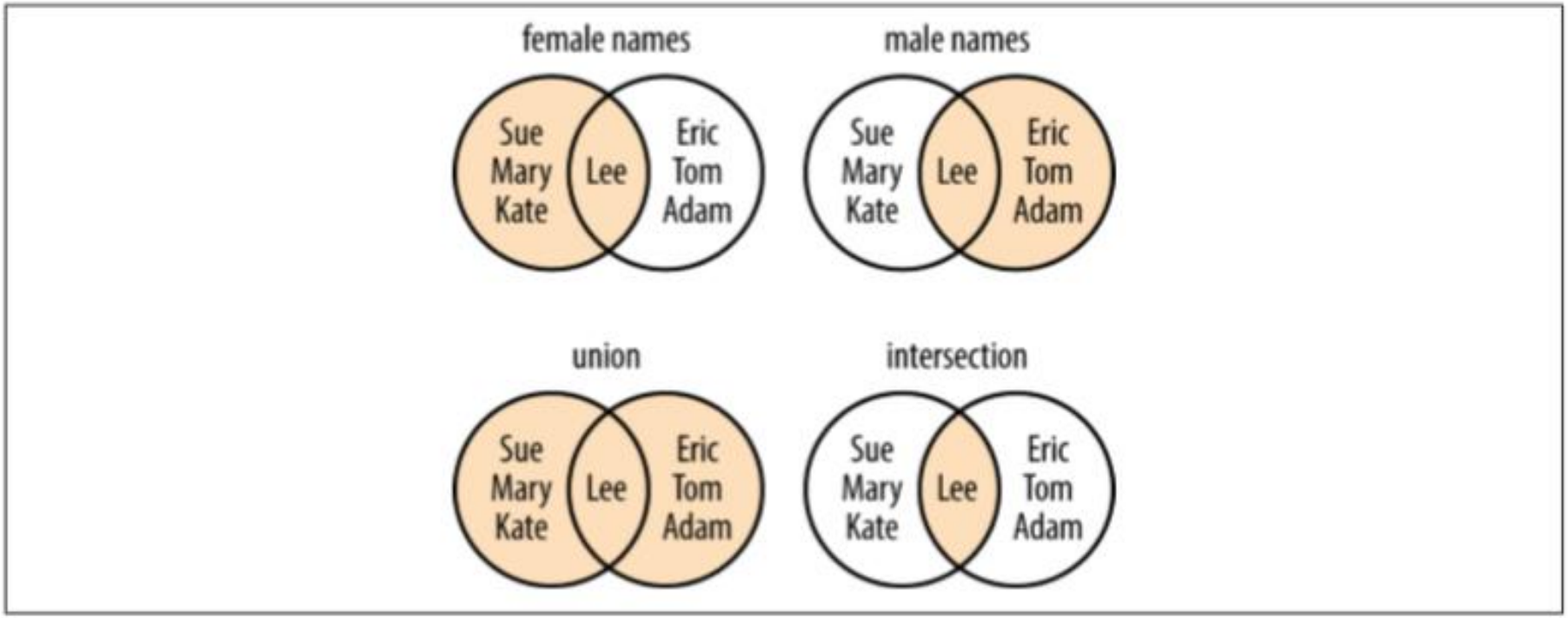
Source: https://xkcd.com/353/



Sets

SET

A set is like a dictionary with its values thrown away, leaving only the keys. As with a dictionary, each key must be unique.



Common things to do with sets

SET

```
>>> empty_set = set()
>>> empty_set
set()
>>> even_numbers = {0, 2, 4, 6, 8}
>>> even_numbers
{0, 8, 2, 4, 6}
>>> odd_numbers = {1, 3, 5, 7, 9}
>>> odd_numbers
{9, 3, 1, 5, 7}
```

SET

```
>>> a = {1, 2}
>>> b = {2, 3}
```

```
>>> a & b
{2}
>>> a.intersection(b)
{2}
>>> a | b
{1, 2, 3}
>>> a.union(b)
{1, 2, 3}

>>> a - b
{1}
>>> a.difference(b)
 {1}
```

SET

The exclusive or (items in one set or the other, but not both) uses **^** or **symmetric_difference():**

```
>>> a = {1, 2}
>>> b = {2, 3}
```

```
>>> a ^ b
{1, 3}
>>> a.symmetric_difference(b)
{1, 3}
```

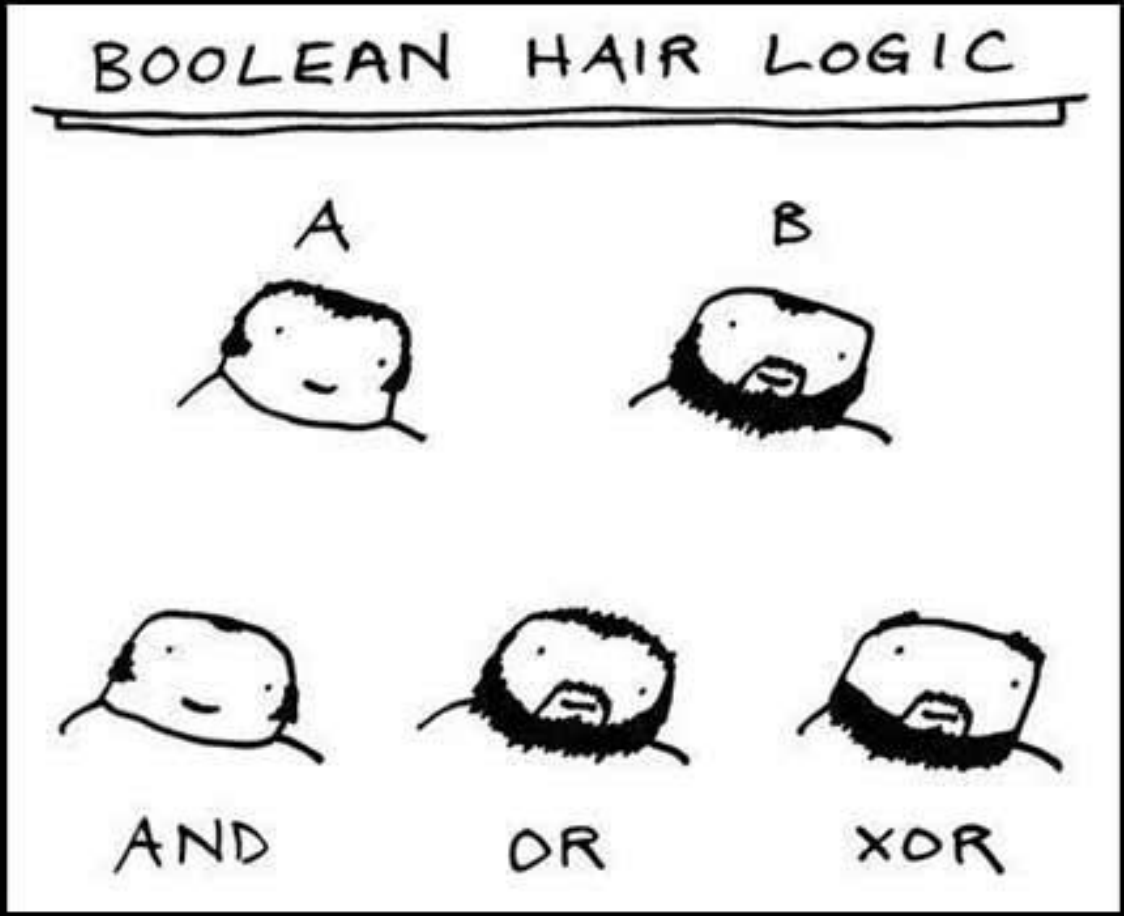
SET

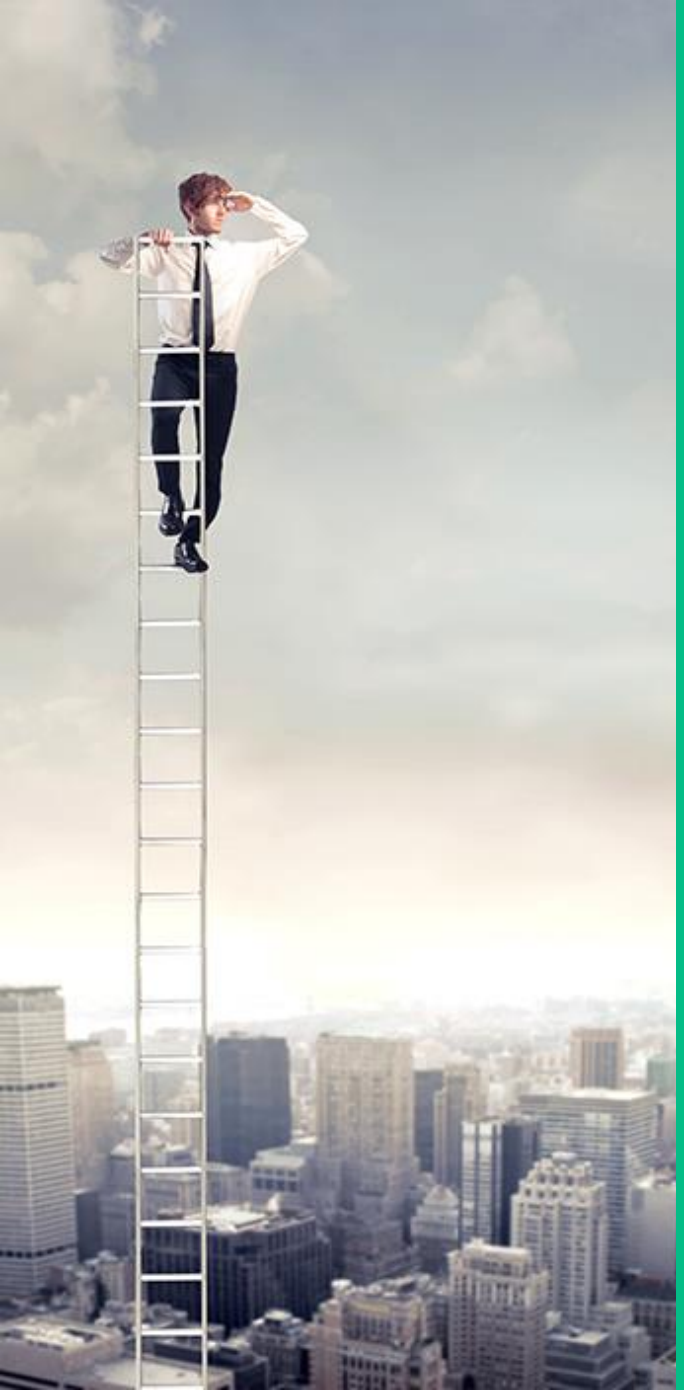
You can check whether one set is a subset of another (all members of the first set are also in the second set) by using <= or issubset():

```
>>> a = {1, 2}
>>> b = {2, 3}
```

```
>>> a <= b
False
>>> a.issubset(b)
False
```

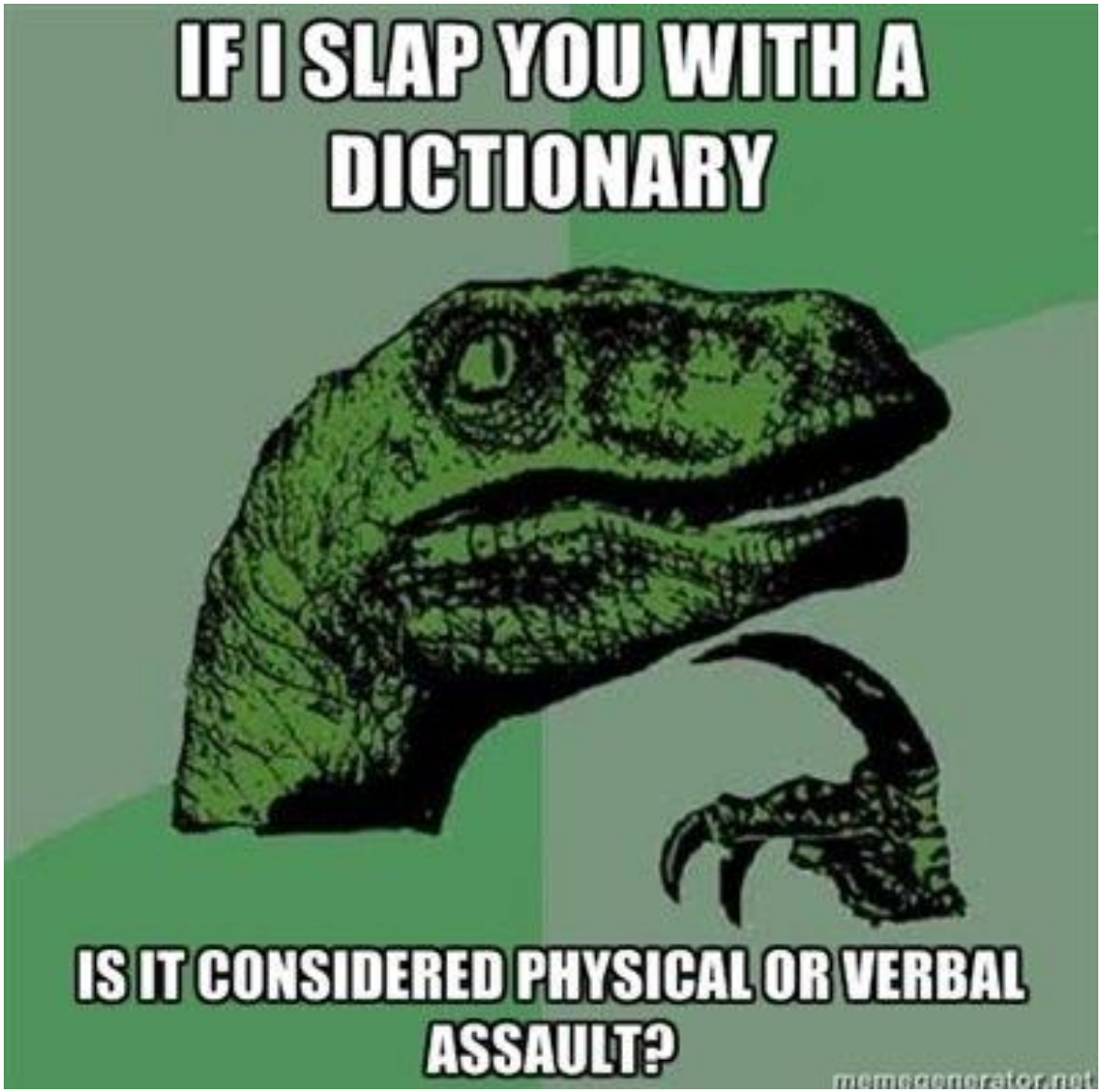
LOGICAL OPERATOR





Dict

DICTS


IF I SLAP YOU WITH A DICTIONARY
IS IT CONSIDERED PHYSICAL OR VERBAL ASSAULT?
memegenerator.net

DICTS

```
>>> empty_dict = {}
>>> empty_dict
{}
```

```
>>> bierce = {
...     "day": "A period of twenty-four hours, mostly misspent",
...     "positive": "Mistaken at the top of one's voice",
...     "misfortune": "The kind of fortune that never misses",
...     }
>>>
```

DICTS

```
osoba = {'imie': 'Jan',
         'nazwisko': 'Kowalski',
         'wzrost': 'normalny'}

print(osoba['imie'])
print(osoba['nazwisko'])

osoba ['wiek'] = 18


del osoba['wzrost']
print (osoba)
```

DICTS

You can use the dict() function to convert two-value sequences into a dictionary.

```
>>> lol = [ ['a', 'b'], ['c', 'd'], ['e', 'f'] ]
>>> dict(lol)
{'c': 'd', 'a': 'b', 'e': 'f'}
```

Order of keys in a dictionary is arbitrary, and might differ depending on how you add items.

DICTS

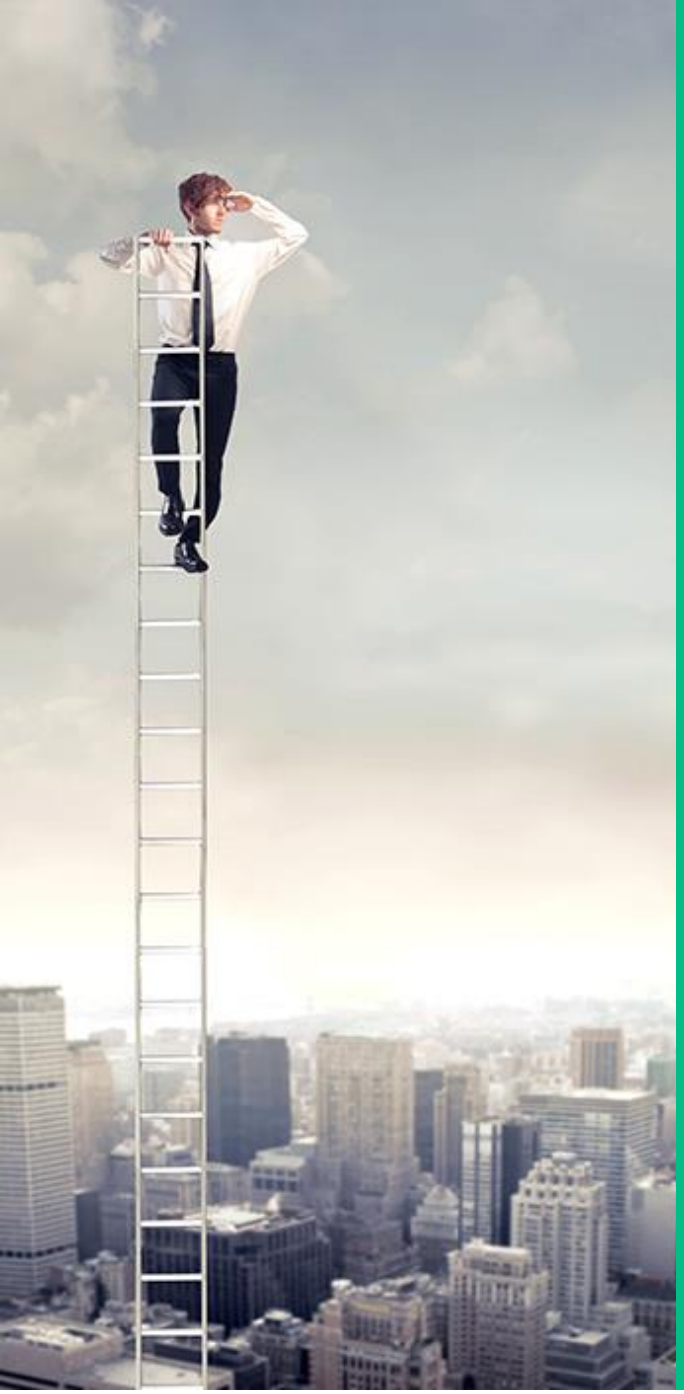
You can use the update() function to copy the keys and values of one dictionary into another.

```
>>> pythons = {
... 'Chapman': 'Graham',
... 'Cleese': 'John',
... 'Gilliam': 'Terry',
... 'Idle': 'Eric',
... }
>>> others = { 'Marx': 'Groucho', 'Howard': 'Moe' }

>>> pythons.update(others)
>>> pythons
{'Cleese': 'John', 'Howard': 'Moe', 'Gilliam': 'Terry',
'Marx': 'Groucho', 'Chapman': 'Graham',
'Idle': 'Eric'}
```

DICTS

You can use keys() to get all the keys in a dictionary.

```
>>> signals = {'green': 'go', 'yellow': 'go
faster', 'red': 'smile for the camera'}

>>> signals.keys()
dict_keys(['green', 'red', 'yellow'])

>>> list( signals.values() )
['go', 'smile for the camera', 'go faster']

>>> list( signals.items() )
[('green', 'go'), ('red', 'smile for the camera'),
('yellow', 'go faster')]
```



Introduction to classes

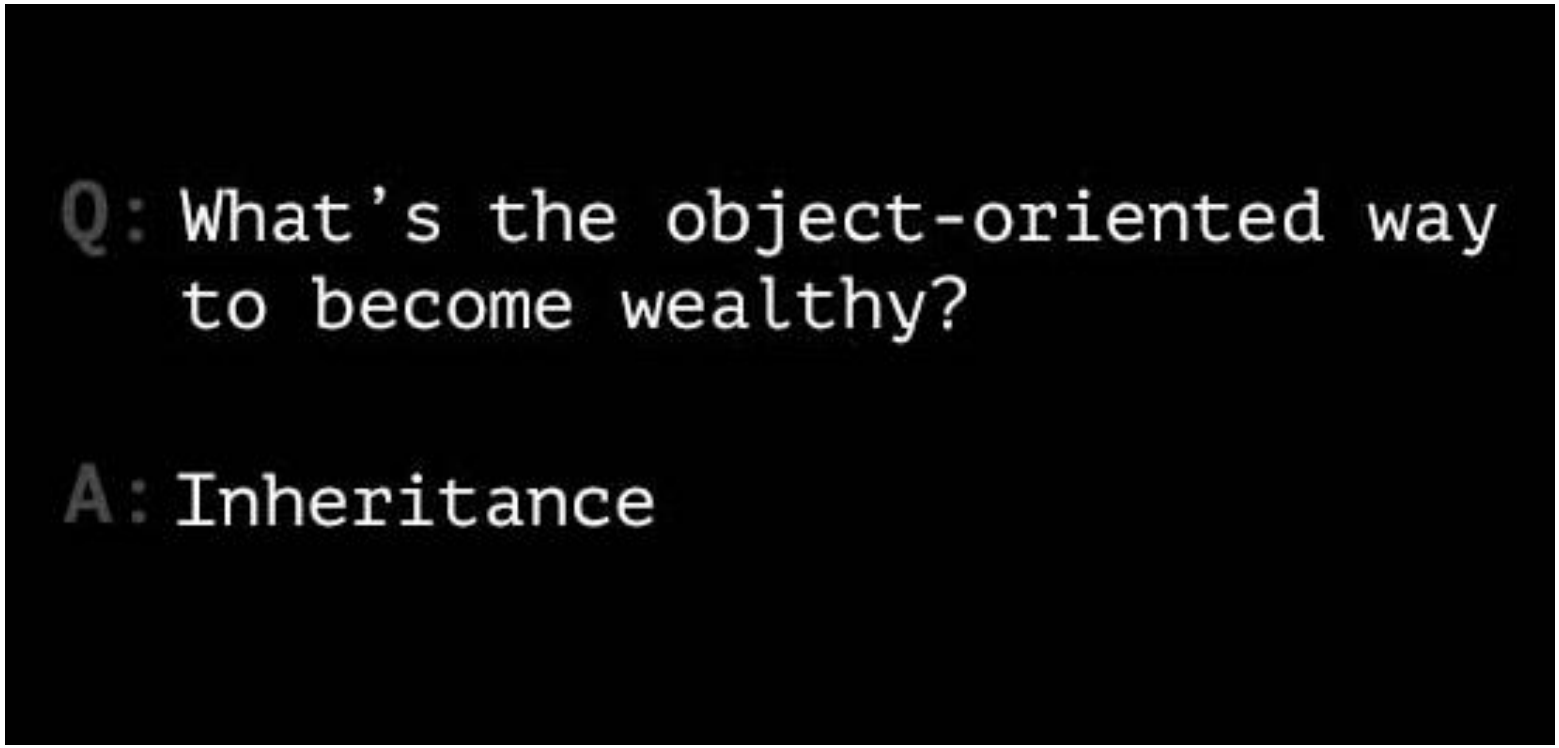
CLASS

```
def this_is_a_normal_function():
    print('I am a normal function')
```

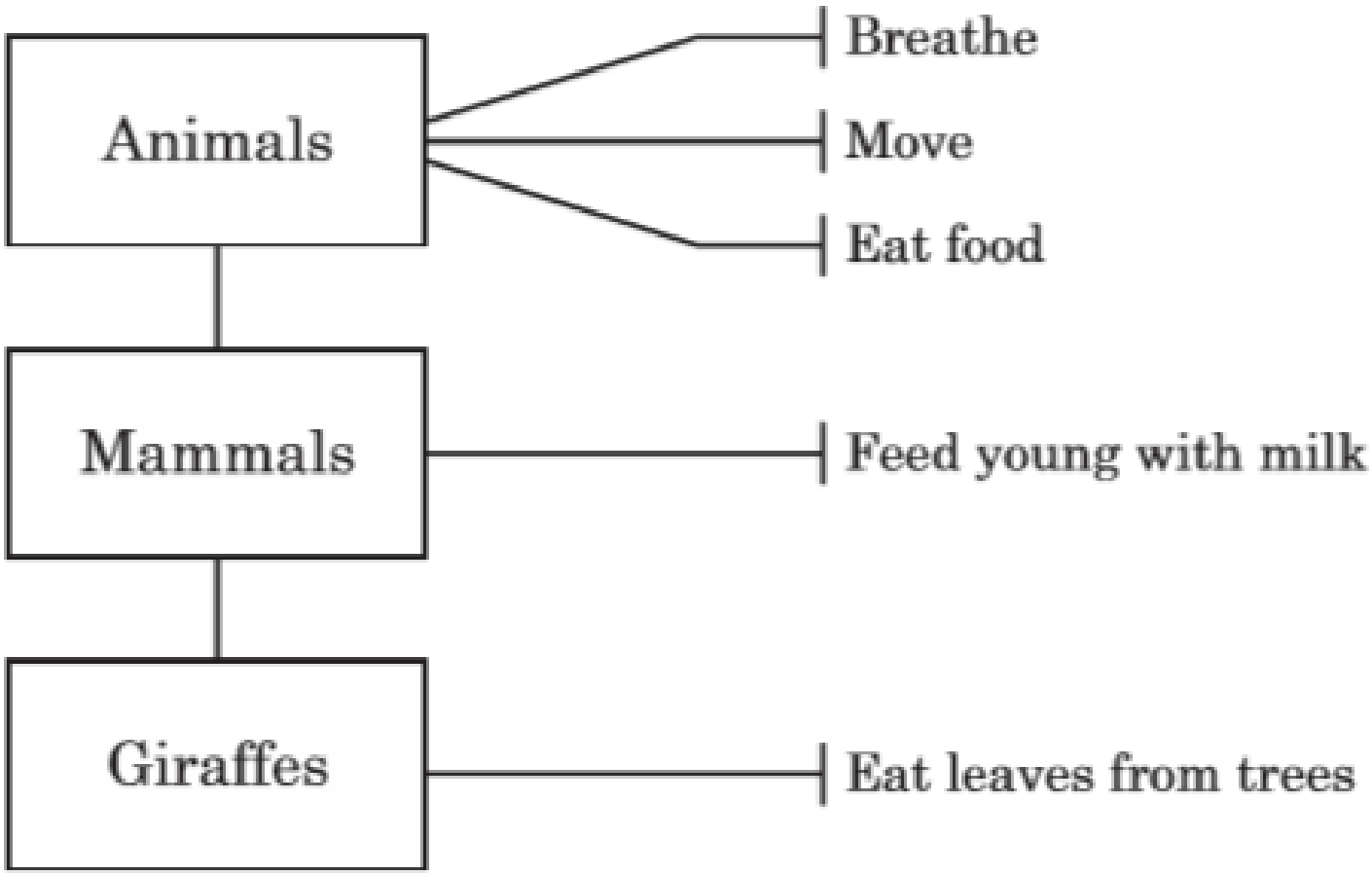
Class example

```
class ThisIsMySillyClass:
    def this_is_a_class_function():
        print('I am a class function')
    def this_is_also_a_class_function():
        print('I am also a class function. See?')
```

JOKE

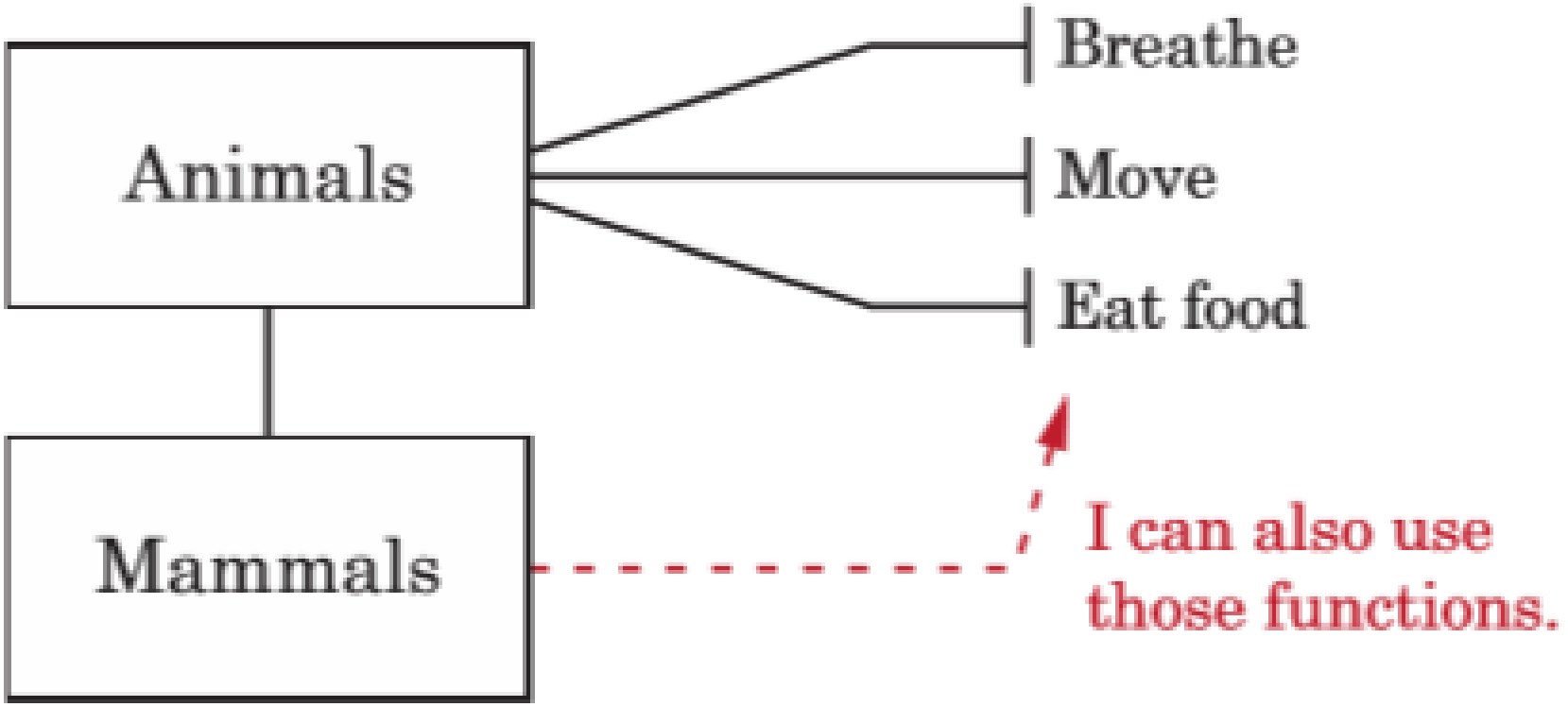

Q: What's the object-oriented way to become wealthy?
A: Inheritance

CLASS



Source: J.R. Briggs, „Python for Kids”

CLASS INHERITANCE



Source: J.R. Briggs, „Python for Kids”

CLASS INHERITANCE

```
class Animals():
    def breathe(self):
        print("oddycham")
    def move(self):
        print("biegam")
    def eat_food(self):
        pass

class Mammals(Animals):
    def feed_young_with_milk():
        print("karmi")

class Giraffes(Mammals):
    def eat_leaves_from_trees(self):
        print('je liscie')

x = Giraffes()
x.eat_leaves_from_trees()
```

Source: J.R. Briggs, „Python for Kids”

CLASS - EXERCISE

Create own animals

CLASS INIT

```
class Song(object):

    def __init__(self, lyrics):
        self.lyrics = lyrics

    def sing_me_a_song(self):
        for line in self.lyrics:
            print line

happy_bday = Song(["Happy birthday to you",
                   "I don't want to get sued",
                   "So I'll stop right there"])

bulls_on_parade = Song(["They rally around the family",
                        "With pockets full of shells"])

happy_bday.sing_me_a_song()

bulls_on_parade.sing_me_a_song()
```

POINT EXAMPLE

```
class Point:
    def __init__(self, x, y):
        self.move(x, y)
    def move(self, x, y):
        self.x = x
        self.y = y
    def reset(self):
        self.move(0, 0)

# Constructing a Point
point = Point(3, 5)
print(point.x, point.y)
```

SUPER()

Get Help from Your Parent with super. How to call parent method from child?

```
>>> class Person():
...     def __init__(self, name):
...         self.name = name
...


>>> class EmailPerson(Person):
...     def __init__(self, name, email):
...         super().__init__(name)
...         self.email = email
```

Source: "Introducing Python. Modern Computing In Simple Packages", B. Lubanovic

In Python everything is an object and has:

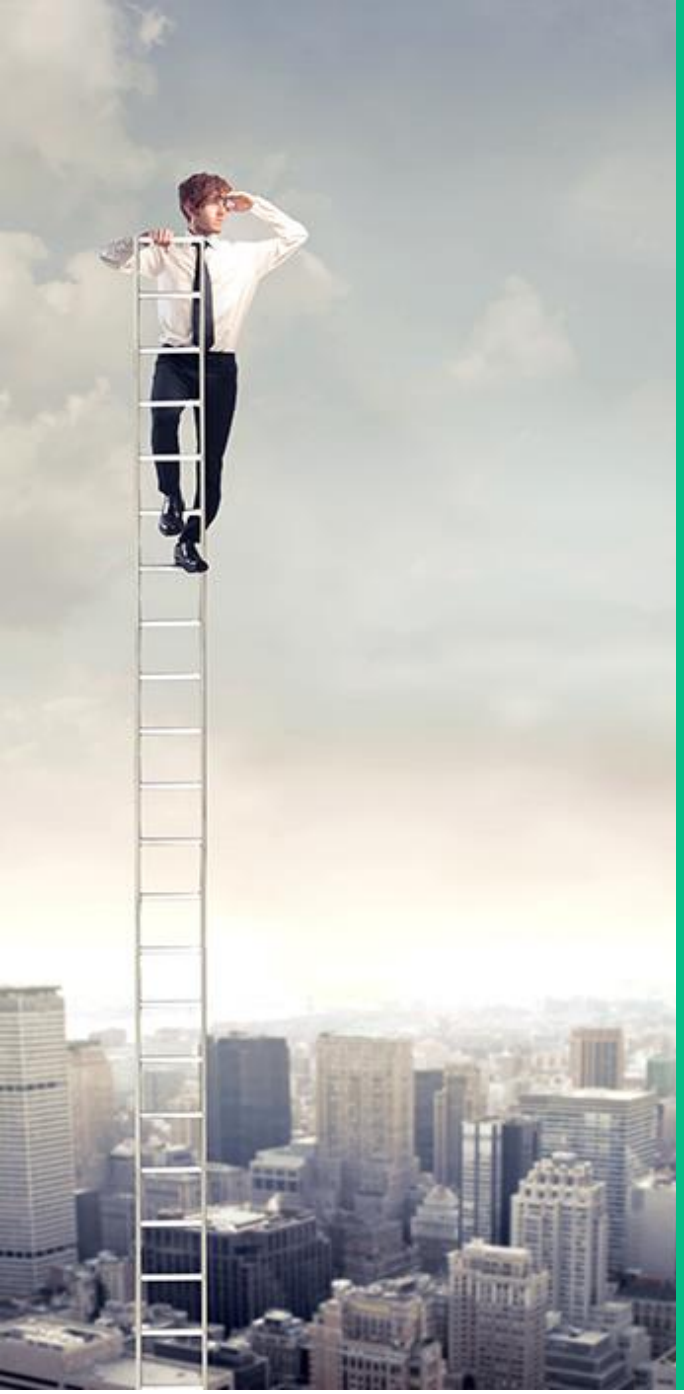
- a single value
- a single type
- some number of attributes
- a single id
- (zero or) one or more names
- and usually one or more base classes

Strong type vs weak type?

PYTHON FEATURE

Python is *strongly typed*, which means that the type of an object does not change, even if its value is mutable

Source: "Introducing Python. Modern Computing In Simple Packages", B. Lubanovic



Using pip and importing external modules

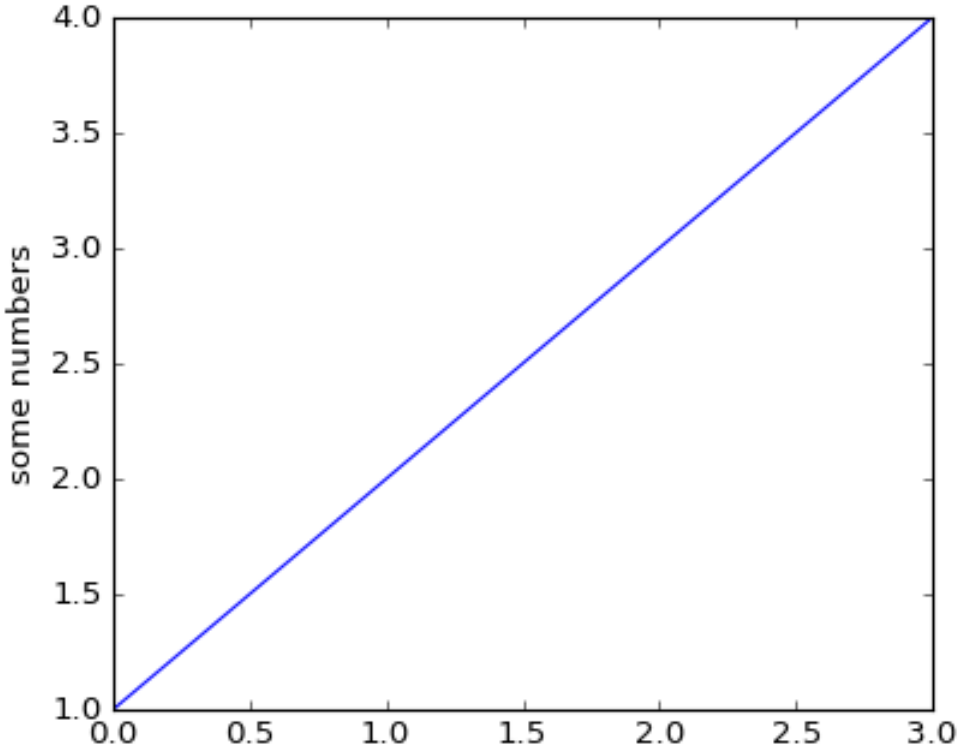
PIP’a

Folder with python/Script/pip.exe

- Gives list of available options

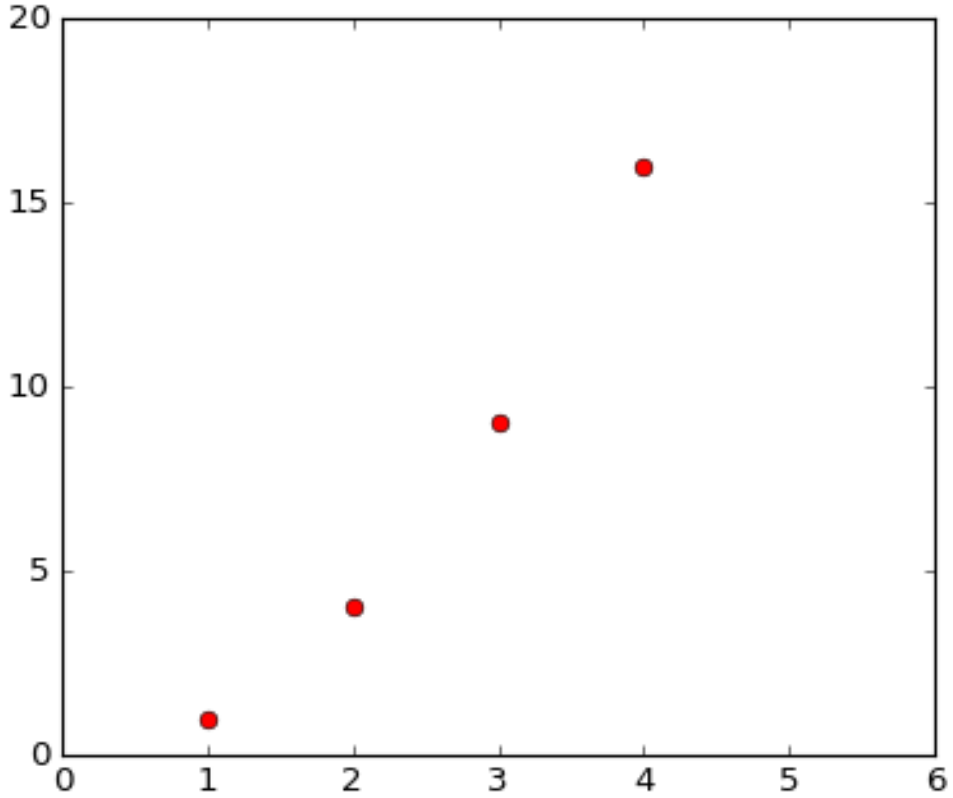
MATPLOTLIB

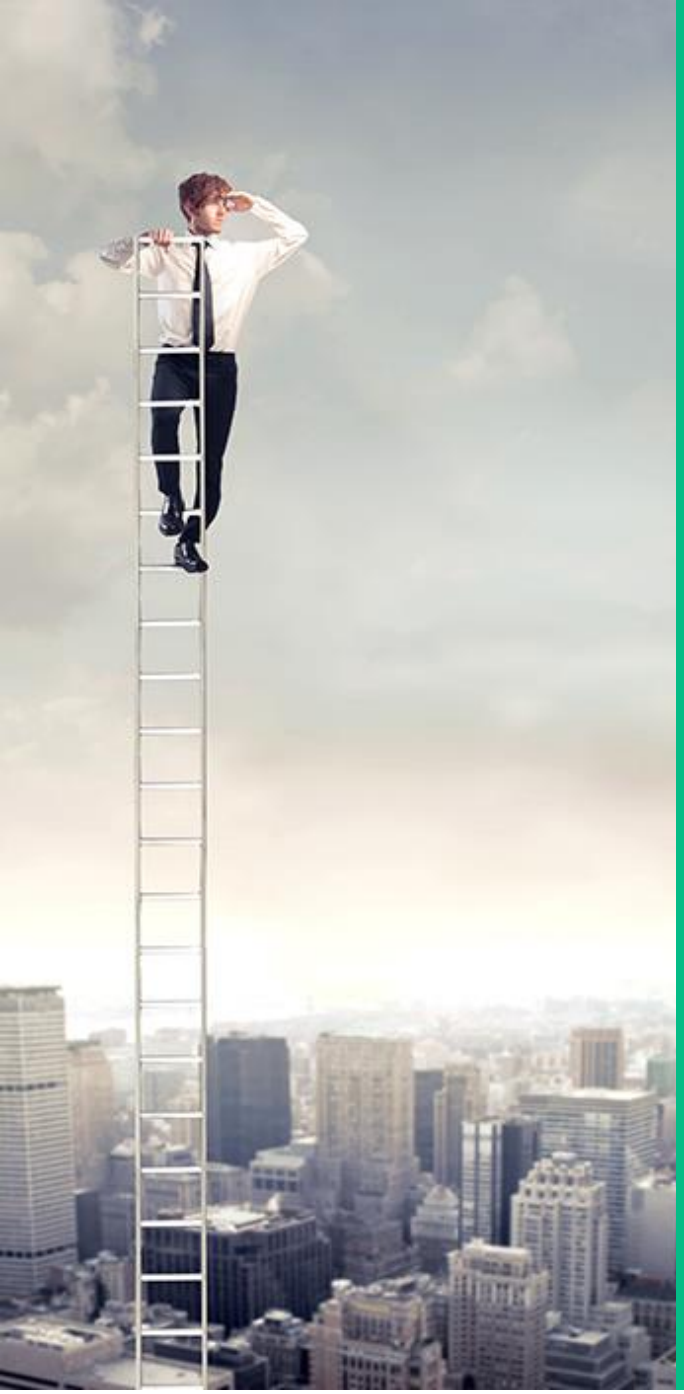
```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4])
plt.ylabel('some numbers')
plt.show()
```



MATPLOTLIB

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```





Program structure

__main__

```
if __name__ == '__ main __’:
```

- to add self-test code
- or to call to a main function at the bottom of module files;

true only when file is run, not when it is imported as a library component.

```
import multiprocessing
import os

def do_this(what):
    whoami(what)

def whoami(what):
    print("Process %s says: %s" % (os.getpid(), what))

if __name__ == "__main__":
    whoami("I'm the main program")
    for n in range(4):
        p = multiprocessing.Process(target=do_this,
          args=("I'm function %s" % n,))
        p.start()
```

Source: "Introducing Python. Modern Computing In Simple Packages", B. Lubanovic

Python\Python35-32\Lib\json\tool.py

```
r"""Command-line tool to validate and pretty-print JSON

Usage::

    $ echo '{"json":"obj"}' | python -m json.tool
    {
        "json": "obj"
    }
    $ echo '{ 1.2:3.4}' | python -m json.tool
    Expecting property name enclosed in double quotes: line 1 column 3 (char 2)

"""
import argparse
import collections
import json
import sys


def main():
    prog = 'python -m json.tool'
    description = ('A simple command line interface for json module '
                   'to validate and pretty-print JSON objects.')
    parser = argparse.ArgumentParser(prog=prog, description=description)
    parser.add_argument('infile', nargs='?', type=argparse.FileType(),
                        help='a JSON file to be validated or pretty-printed')
    parser.add_argument('outfile', nargs='?', type=argparse.FileType('w'),
                        help='write the output of infile to outfile')
    parser.add_argument('--sort-keys', action='store_true', default=False,
                        help='sort the output of dictionaries alphabetically by key')
    options = parser.parse_args()

    infile = options.infile or sys.stdin
    outfile = options.outfile or sys.stdout
    sort_keys = options.sort_keys
    with infile:
        try:
            if sort_keys:
                obj = json.load(infile)
            else:
                obj = json.load(infile,
                                object_pairs_hook=collections.OrderedDict)
        except ValueError as e:
            raise SystemExit(e)
    with outfile:
        json.dump(obj, outfile, sort_keys=sort_keys, indent=4)
        outfile.write('\n')


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

PEPS

PEP 0 -- Index of Python Enhancement Proposals (PEPs)

PEP:	0
Title:	Index of Python Enhancement Proposals (PEPs)
Last-Modified:	2018-05-10
Author:	David Goodger <goodger at python.org>, Barry Warsaw <barry at python.org>
Status:	Active
Type:	Informational
Created:	13-Jul-2000

Introduction

```
This PEP contains the index of all Python Enhancement Proposals,
known as PEPs.  PEP numbers are assigned by the PEP editors, and
once assigned are never changed[1].  The version control history[2] of
the PEP texts represent their historical record.
```

Source: https://www.python.org/dev/peps/

PEP 8 -- Style Guide for Python Code

PEP:	8
Title:	Style Guide for Python Code
Author:	Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org>, Nick Coghlan <ncoghlan at gmail.com>
Status:	Active
Type:	Process
Created:	05-Jul-2001
Post-History:	05-Jul-2001, 01-Aug-2013

Source: https://www.python.org/dev/peps/pep-0008/

STATIC ANALYSE WITH PYLINT

Debian

```
sudo apt-get install pylint
```

Ubuntu

```
sudo apt-get install pylint
```

Fedora

```
sudo dnf install pylint
```

Gentoo

```
emerge pylint
```

openSUSE

```
sudo zypper install pylint  # python2.7
sudo zypper install python3-pylint
```

FreeBSD

```
portmaster devel/pylint
```

Arch Linux

```
pacman -S python2-pylint # if you live in the past
pacman -S python-pylint  # if you live in the future
```

OS X

```
pip install pylint
```

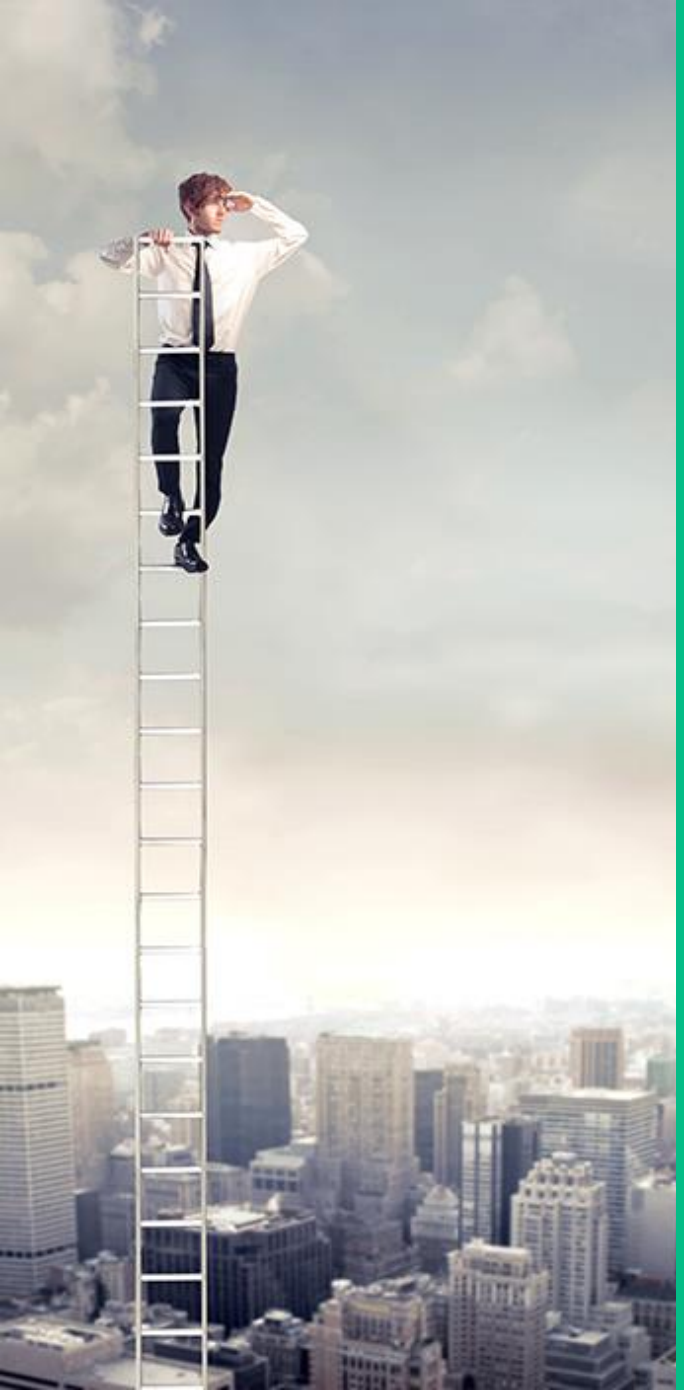
Windows

```
pip install pylint # see note
```

From source using Git

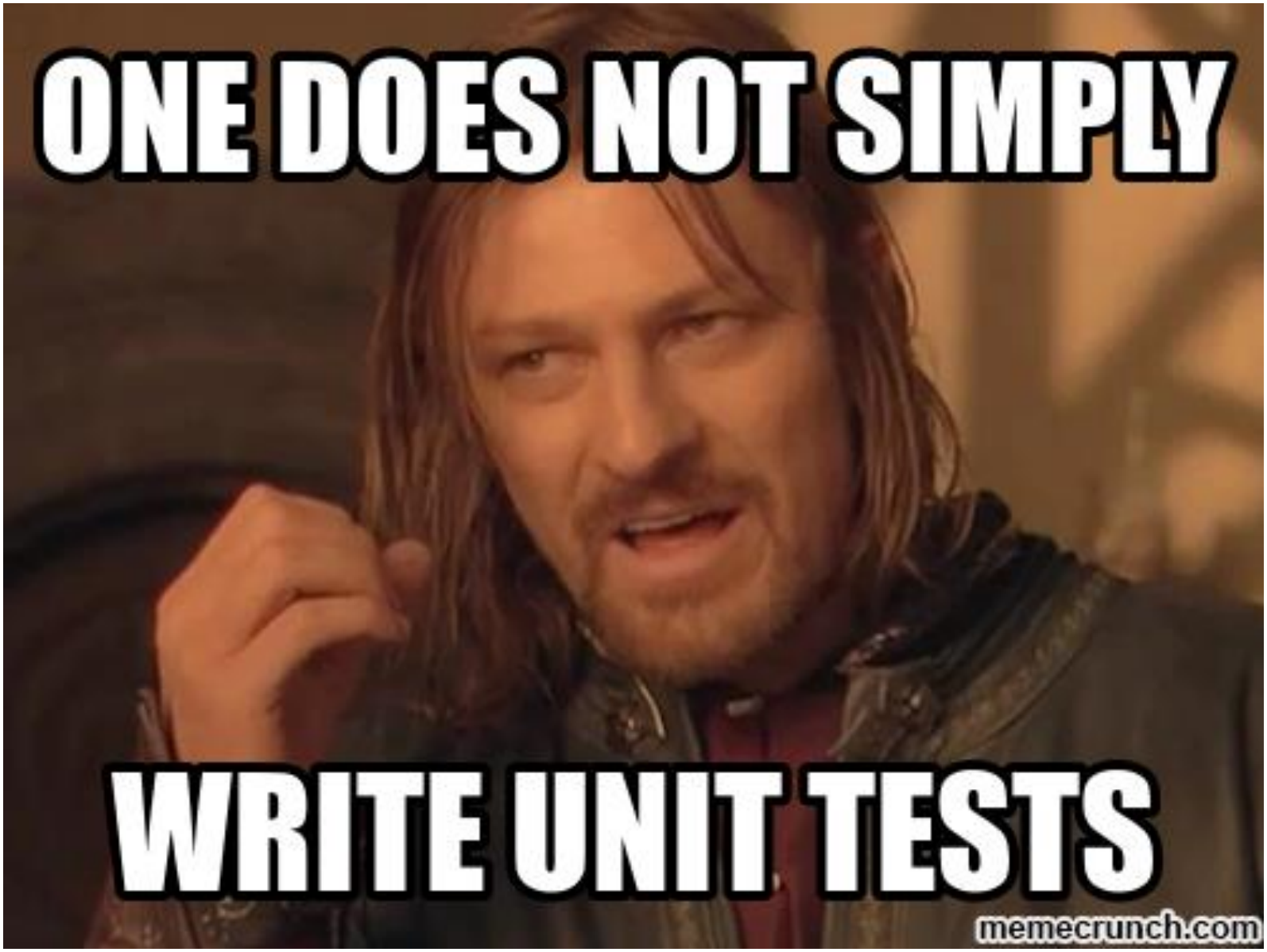
```
git clone https://github.com/PyCQA/pylint
git clone https://github.com/PyCQA/astroid
```

Source: https://www.pylint.org/#install

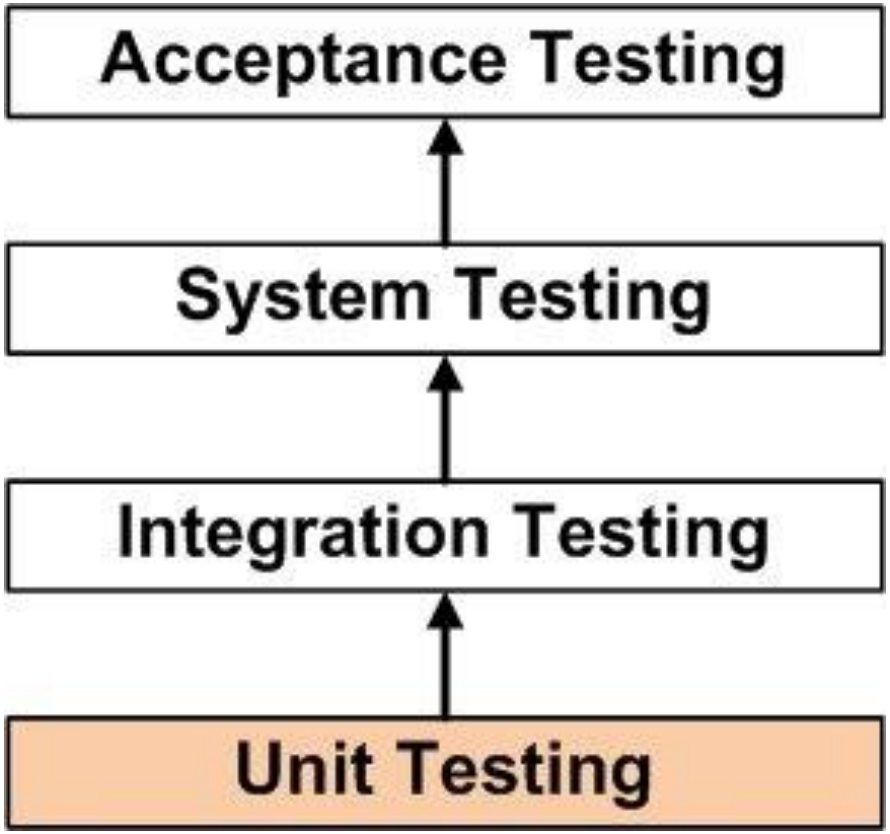


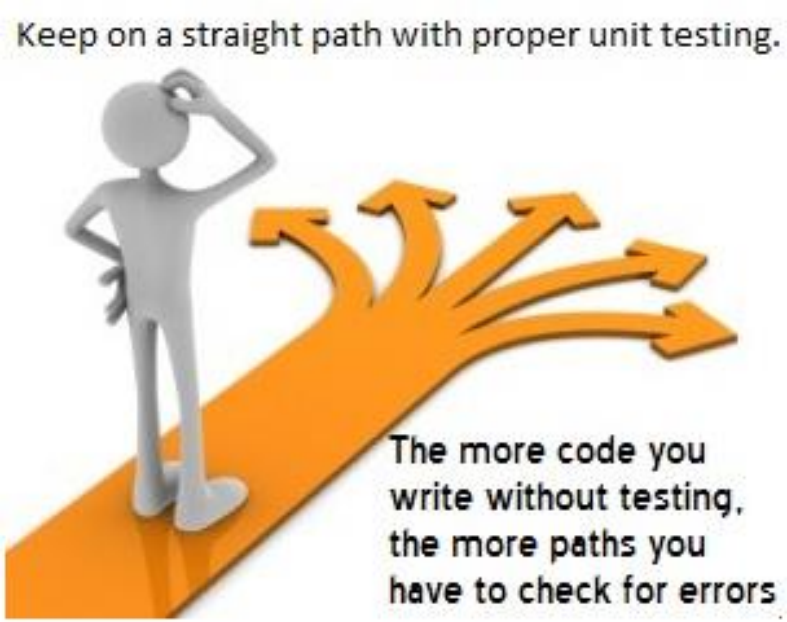
Unit Tests

Unit Tests



Unit Tests


Acceptance Testing
System Testing
Integration Testing
Unit Testing


Keep on a straight path with proper unit testing.
The more code you write without testing, the more paths you have to check for errors


TEST

Unit Tests

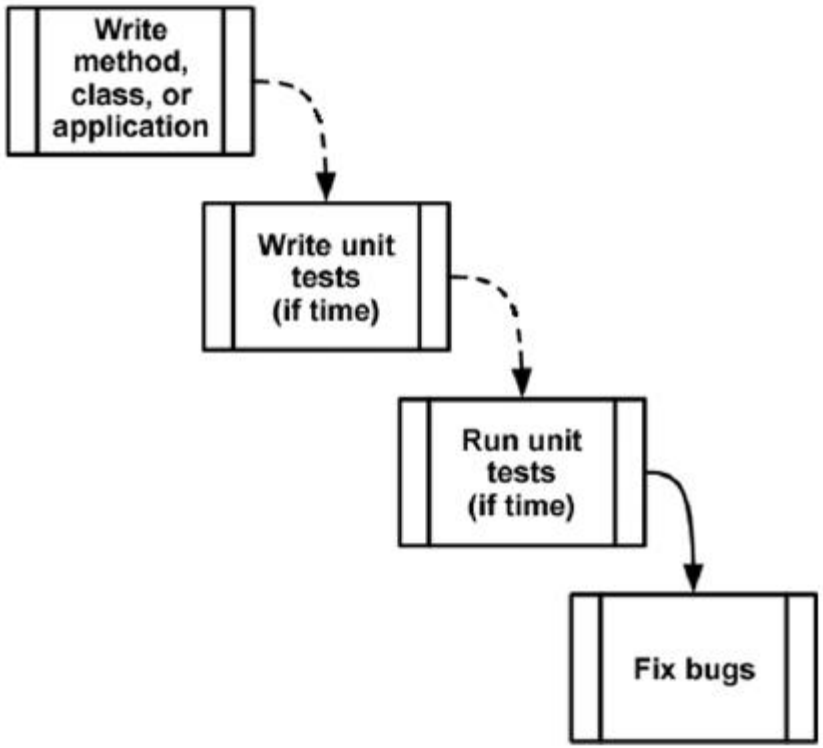
A unit test *should* have the following properties:

- be automated and repeatable.
- easy to implement.
- be relevant tomorrow.
- it should run quickly.
- have full control of the unit under test.
- Anyone should be able to run it at the push of a button.
- It should be consistent in its results (it always returns the same result if you don't change anything between runs)
- fully isolated (runs independently of other tests).
- when it fails, it should be easy to detect what was expected and determine how to pinpoint the problem.

Source: Roy Osherove - The Art of Unit Testing, Second Edition

Unit Tests

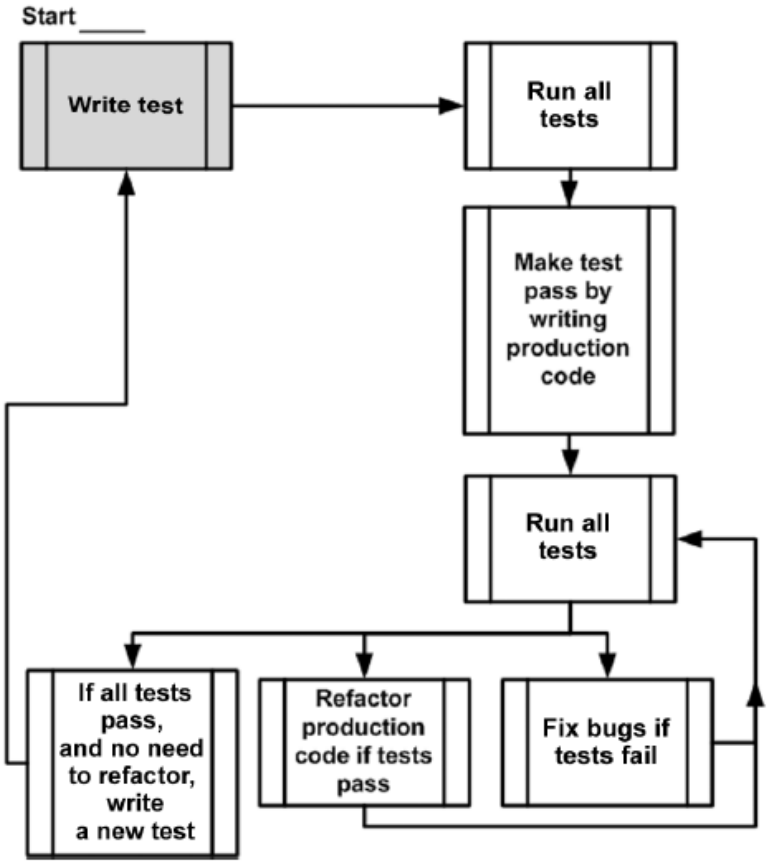
- The traditional way of writing unit tests.
- The broken lines represent actions people treat as optional.



Source: Roy Osherove - The Art of Unit Testing, Second Edition

Unit Tests

- Test-driven development. Notice the spiral nature of the process: write test, writebcode, refactor, write next test. It shows the incremental nature of TDD: small steps lead to a quality end result.

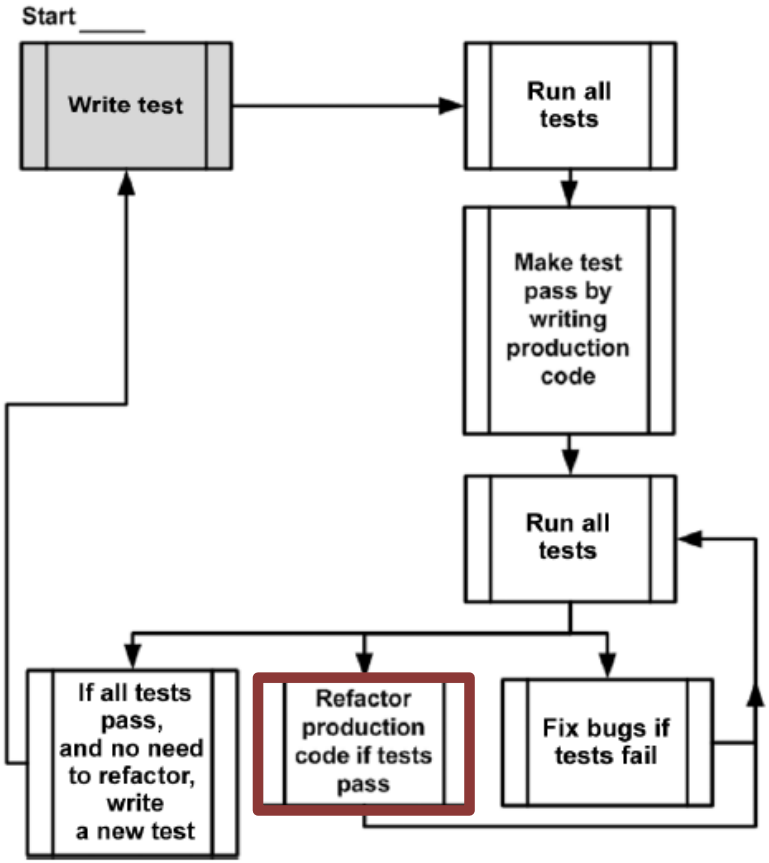


Source: Roy Osherove - The Art of Unit Testing, Second Edition

192

Unit Tests

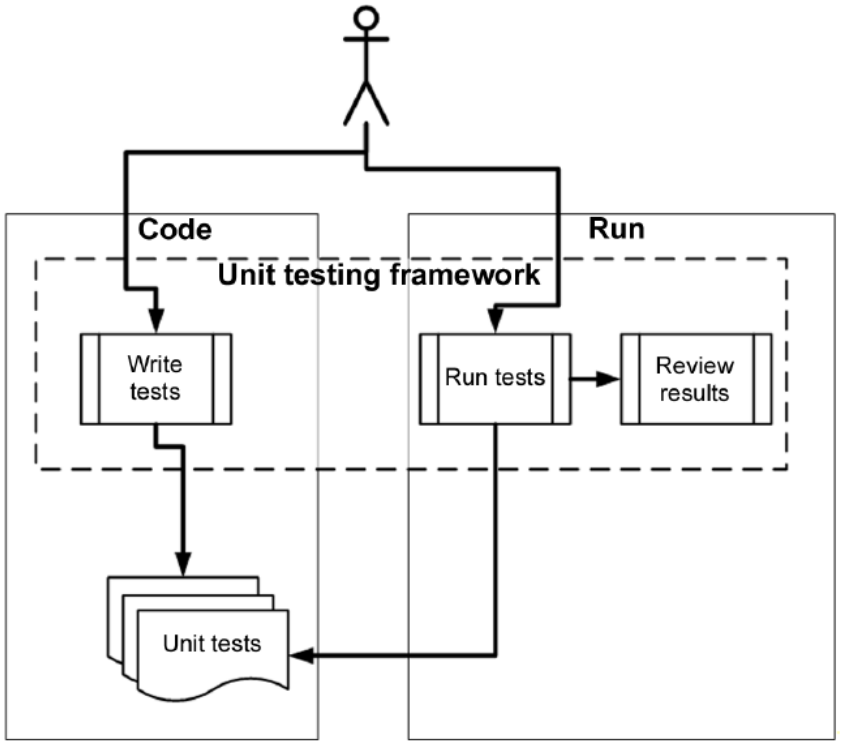
- *Refactoring* means changing a piece of code *without* changing its functionality.
- If you've ever renamed a method, you've done refactoring. If you've ever split a large method into multiple smaller method calls, you've refactored your code. The code still does the same thing, but it becomes easier to maintain, read, debug, and change.



Source: Roy Osherove - The Art of Unit Testing, Second Edition

Unit Tests

- Unit tests are written as code, using libraries from the unit testing framework. Then the tests are run from a separate unit testing tool or inside the IDE, and the results are reviewed (either as output text, the IDE, or the unit testing framework application UI) by the developer or a



Source: Roy Osherove - The Art of Unit Testing, Second Edition

Unit Tests

```
class Calculate(object):
    def add(self, x, y):
        return x + y

if __name__ == '__main__':
    calc = Calculate()
    result = calc.add(2, 2)
    print result
```

Source: David Sale - Testing Python Applying Unit Testing, TDD, BDD And Acceptance Testing

Unit Tests

```
import unittest
from app.calculate import Calculate


class TestCalculate(unittest.TestCase):
    def setUp(self):
        self.calc = Calculate()
    def test_add_method_returns_correct_result(self):
        self.assertEqual(4, self.calc.add(2,2))


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

```
$ python test/calculate_test.py
.
----------------------------------
  ---
Ran 1 test in 0.000s
OK
```

Source: David Sale - Testing Python Applying Unit Testing, TDD, BDD And Acceptance Testing

Useful Methods in Unit Testing

- `assertEqual(x, y, msg=None)`
- `assertAlmostEqual(x, y, places=None, msg=None, delta=None)`
- `assertRaises(exception, method, arguments, msg=None`
- `assertDictContainsSubset(expected, actual, msg=None)`
- `assertDictEqual(d1, d2, msg=None)`
- `assertTrue(expr, msg=None)`
- `assertFalse(expr, msg=None)`
- `assertGreater(a, b, msg=None)`
- `assertGreaterEqual(a, b, msg=None`
- `assertIn(member, container, msg=None)`

Source: David Sale - Testing Python Applying Unit Testing, TDD, BDD And Acceptance Testing

Useful Methods in Unit Testing

- `assertIs(expr1, expr2)`
- `assertIsInstance(obj, class, msg=None)`
- `assertNotIsInstance(obj, class, msg=None)`
- `assertIsNone(obj, msg=None)`
- `assertIsNot(expr1, expr2, msg=None)`
- `assertIsNotNone(obj, msg=None)`
- `assertLess(a, b, msg=None)`
- `assertLessEqual(a, b, msg=None)`
- `assertItemsEqual(a, b, msg=None)`
- `assertRaises(excClass, callableObj, *args, **kwargs, msg=None)`

Source: David Sale - Testing Python Applying Unit Testing, TDD, BDD And Acceptance Testing