Basics

History

- 1989: **Guido van Rossum** (At CWI, Netherland)
- 1991: First public release
- 1995: CNRI, USA
- 1999: Computer Programming for Everybody (CNRI + DARPA)
  *Objective : python for **teaching** programming*
- 2001: Python software fundation (free from governments)
- 2008: version 3
- 2013: widely use in post-graduate
- 2018: 4th most popular language (behind Java, C and C++)
- 2020: death of python 2.x?

Advantages of Python

- **Batteries included** (+numpy, scipy, scikits)
- **Easy to learn**
- **Easy communication**
- **Efficient code**
- **Universal**

Install

- Windows : anaconda + conda + pip
- Linux : already install + apt + pip --user
- Mac : homebrew + brew python3 + pip --user

Interface

vim+terminal / pycharm / spyder / jupyter / python

First examples

Helloworld

In [1]:

```
print("Hello World!")
```

```
Hello World!
```

Arithmetic

In [2]:

```
print(2 + 3 * 9 / 6)
```

```
6.5
```

****As simple as that!****

Vocabulary

- `mutable` / `immutable`
  `unashable` / `hashable`
  a mutable object can be altered an immutable object can not
  obviously numerical object are immutable
- `exception`
  wathever you do, python never crash (almost never)
  Errors detected during execution are called exceptions and you can handle them

In [3]:

```
def test():
    error = {[0]:4}
test()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-219e8c1ed0c0> in <module>()
      1 def test():
      2     error = {[0]:4}
----> 3 test()

<ipython-input-3-219e8c1ed0c0> in test()
      1 def test():
----> 2     error = {[0]:4}
      3 test()

TypeError: unhashable type: 'list'
```

Help

- help
- dir
- internet
    - docs.python.org
    - stackoverflow
    - reddit
    - ...

Syntax

- Commments
    - use hash symbol (#) to start a comment

- Variables:
    - name (identifier)
        - combination of letters or digits or an underscore (_)
        - cannot start with a digit
        - case sensitive
        - no keywords
        - no operator symbol
        - accented caracter allowed (but to avoid)
    - dynamic type
        - `None`
        - numericals type
            - integer infinite precision
            - float double precision
            - complex
        - iterables type
            - tuple
            - list
            - set
            - dist
            - str
            - frozenset
            - bytarray

- operators
  `+ - * ** / // % = < > <= >= == != () {} [] ‘ " @ . ~ & |`

- keywords:

In [4]:

```
from keyword import kwlist
print(kwlist)
```

```
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class',
'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'fo
r', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonloc
al', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'wit
h', 'yield']
```

Simple exemple

In [5]:

```
# A comment
a_variable = 1
another_variable = 1.2
another_Variable = a_variable + another_variable
print(a_variable, another_variable, another_Variable)

a_variable = "spam "
a_variable2 = "egg"
print(a_variable * 2 + a_variable2)

print(10 ** 180 + 1)
```

```
1 1.2 2.2
spam spam egg
1000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000001
```

First exercise

From a radius R and a length `l`, compute the perimetre, the surface and the volume of a cylinder

In [6]:

```
R = 0.7
l = 0.1
pi = 3.14

# TODO
```

Solution

In [7]:

```
perimetre = 2 * (2 * pi * R)
surface =  2 * pi * R * (R + l)
volume = l * pi * R ** 2

print("Perimetre = ", perimetre)
print("Surface = ", surface)
print("Volume = ", volume)
```

```
Perimetre =  8.792
Surface =  3.5167999999999995
Volume =  0.15386
```

Code blocks

```
code block:
    stuff inside the code block
    more stuff  inside the code block
    pass # do nothing
outside of the code block
```

- tests : `if test`
- functions : `def functionname()`
- loops : `for variable in iterable`, `while test`
- objects : `class objectname()`
- contexts : `with context`
- exception handling : `try`, `except`

`def` Statements

```
def a_function(arg, optionnal_arg=default_value):
    #...
    return something
```

In [8]:

```
def quadcube(x):            # define a function with argument
    return x ** 2, x ** 3  # multiple returns

x1, x2 = quadcube(7)        # multiple assignment
print(x1, x2)
```

```
49 343
```

`if` Statements

```
if test:
    action
else:
    other_action
```

Equivalent forms of True:

- `True`
- any non-zero value
- an non-empty iterable

Equivalent forms of False:

- `False`
- `None`
- `0`
- an empty iterable

For tests, you can use those operators:

```
and, or, not, ==, !=, is, is not, >, >=, <, <=, in, not in
```

In [9]:

```
def factorial(n):
    if n < 2:                           # define a test
        return 1                        # value return
    else:
        return n * factorial(n - 1)     # recursion

res = factorial(30)                     # call the function
print(res)
```

```
265252859812191058636308480000000
```

In [10]:

```
def factorial_six():                 # function without argument
    return factorial(6)

res = factorial_six()
print(res)
```

```
720
```

while loop

```
i = 0
while i < 4:
    ...
    i += 1
```

Loop over iterable object

```
for i in [5, 3, 4]:
    ...
```

Usualy, we use `range(start=0, end, step=1)`
which kind of create `[start, start + step, ..., end - step]`

WARNING

\`end\` is the first value **never** reached

In [11]:

```
def is_prime(num):
    if num == 1:
        return False
    i = 2
    while i < int(num ** 0.5) + 1:
        if (num % i) == 0:
            return False
        i += 1
    return True

def is_prime2(num):                             # equivalant variation
    if num == 1:
        return False
    for i in range(2, int(num ** 0.5) + 1):  # using range
        if (num % i) == 0:
            return False
    return True

print(is_prime(7))                   # chain functions (f(g(...)))
print(is_prime2(13))
```

```
True
True
```

In [12]:

```python
def compute_pi(err, nmax=float("inf")):    #optional argument
    n = 0
    error = float("inf")

    a_n = 1.
    b_n = 2 ** -0.5
    t = 0.25
    while error > err and n < nmax:
        a_np = 0.5 * (a_n + b_n)
        b_np = (a_n * b_n) ** 0.5
        t -= (2 ** n) * (a_n - a_np) ** 2  # inplace substraction

        a_n, b_n = a_np, b_np              # double assigment

        error = abs(a_n - b_n)
        n += 1                             # inplace addition

    pi = (a_n + b_n) ** 2 / (4 * t)
    return pi, n, error

print(compute_pi(1e-15))
print(compute_pi(1e-15, 3))
print(compute_pi(1e-15, nmax=2))
```

```
(3.141592653589794, 4, 1.1102230246251565e-16)
(3.141592653589794, 3, 8.242750926257258e-11)
(3.141592646213543, 2, 2.3636176602614967e-05)
```

Second exercise

A pythagorean triplet is a triplet of positive integers a, b and c such that $a^2 + b^2 = c^2$.

For each pair of positive integers m, n:

- a $= m^2 - n^2$
- b $= 2mn$
- c $= m^2 + n^2$
  is a pythagorean triplet if and only if a, b and c are strictly positive.

Write a function `pythagorean_triplet(limit)` who:

1. loops on n and m
2. use the previous property in order to detect a triplet
3. print all the triplet until a limit, i.e. $c < limit$.

In [13]:

```
# Exercise
def pythagorean_triplets(limit):
    """
    Print all pythagorean triplets below a limit

    A pythagorean triplet is a triplet of integers a, b and c such that
    a^2 + b^2 = c^2

    My solution :
    3 4 5
    8 6 10
    5 12 13
    15 8 17
    12 16 20
    7 24 25
    24 10 26
    21 20 29

    """                                       # use a docstring

help(pythagorean_triplets)
pythagorean_triplets(30)
```

```
Help on function pythagorean_triplets in module __main__:

pythagorean_triplets(limit)
    Print all pythagorean triplets below a limit

    A pythagorean triplet is a triplet of integers a, b and c such
that
    a^2 + b^2 = c^2

    My solution :
    3 4 5
    8 6 10
    5 12 13
    15 8 17
    12 16 20
    7 24 25
    24 10 26
    21 20 29
```

Solution

In [14]:

```python
def pythagorean_triplets(limit):
    """
    Print all pythagorean triplets below a limit

    A pythagorean triplet is a triplet of integers a, b and c such that
    a^2 + b^2 = c^2
    """                                   # use a docstring

    ## compute range of m ##

    # m ** 2 cannot be bigger than limit
    m_max = limit ** 0.5

    # m_max must be an integer
    m_max = int(m_max)

    # m will be 1, 2, ..., m_max
    for m in range(1, m_max + 1):

        ## compute range of n ##

        # n cannot be bigger than m
        # otherwise a would be negative
        n_max_a = m - 1

        # n cannot be bigger than nmax_c
        # otherwise c would be above the limit
        n_max_c = (limit - m * m) ** 0.5

        n_max = min(n_max_a, n_max_c)

        # n_max must be an integer
        n_max = int(n_max)

        # n will be 1, 2, ..., n_max
        for n in range(1, n_max + 1):

            # compute the triplets
            a = m * m - n * n
            b = 2 * m * n
            c = m * m + n * n

            # print result
            print(a, b, c)
pythagorean_triplets(30)
```

```
3 4 5
8 6 10
5 12 13
15 8 17
12 16 20
7 24 25
24 10 26
21 20 29
```

Lists, dicts and others iterables

- Common types
    - **list** : `[a, b, c]`
        - mutable
        - indexable with integer
        - initialization :
            - `l = [0] * n`
            - `l = [0 for i in range(n)]`
        - operators
            - concatenation : +
            - repetition : *
    - str : `"spam" == 'spam' == """spam"""`
        - immutable
        - indexable with integer
        - can contains only decoded characters
        - unicode par default
        - operators
            - concatenation : +
            - repetition : *
    - tuple : `(a, b, c)`
        - immutable
        - indexable with integer
        - parentheses optional (`a, b = c, d`)
    - set : `{a, b, c}`
        - mutable
        - indexable with integer
        - no duplicate
        - can contains only immutables
    - dict : `{a:b, c:d}`
        - mutable
        - indexable with keys (here a and c)
        - keys are unique and of immutable type
        - no need to initialize before registering a value

- Uncommon types
    - frozenset
        - immutable
        - indexable with integer
        - no duplicate
        - can contains only immutables
    - bytearray
        - mutable
        - indexable with integer
        - can contains only encoded characters

- Slicing
    - `A[1]` : 2nd item
    - `A[-1]` : last item
    - `A[4:8]` : sublist
    - `A[4:8:2]` : sublist by step of 2 (i.e. 5th and 7th item)

- Methods
    - `len`
    - `append`
    - `sort` / `sorted`
    - ...

- Loop on iterable object `for i in [a, b, c]:`

- Comprehension `[x**2 for x in range(10)]`

- Generators / Iterator
    - `range`
    - `enumerate`
    - `zip`
    - `open("filename")`
    - ...

An example

In [15]:

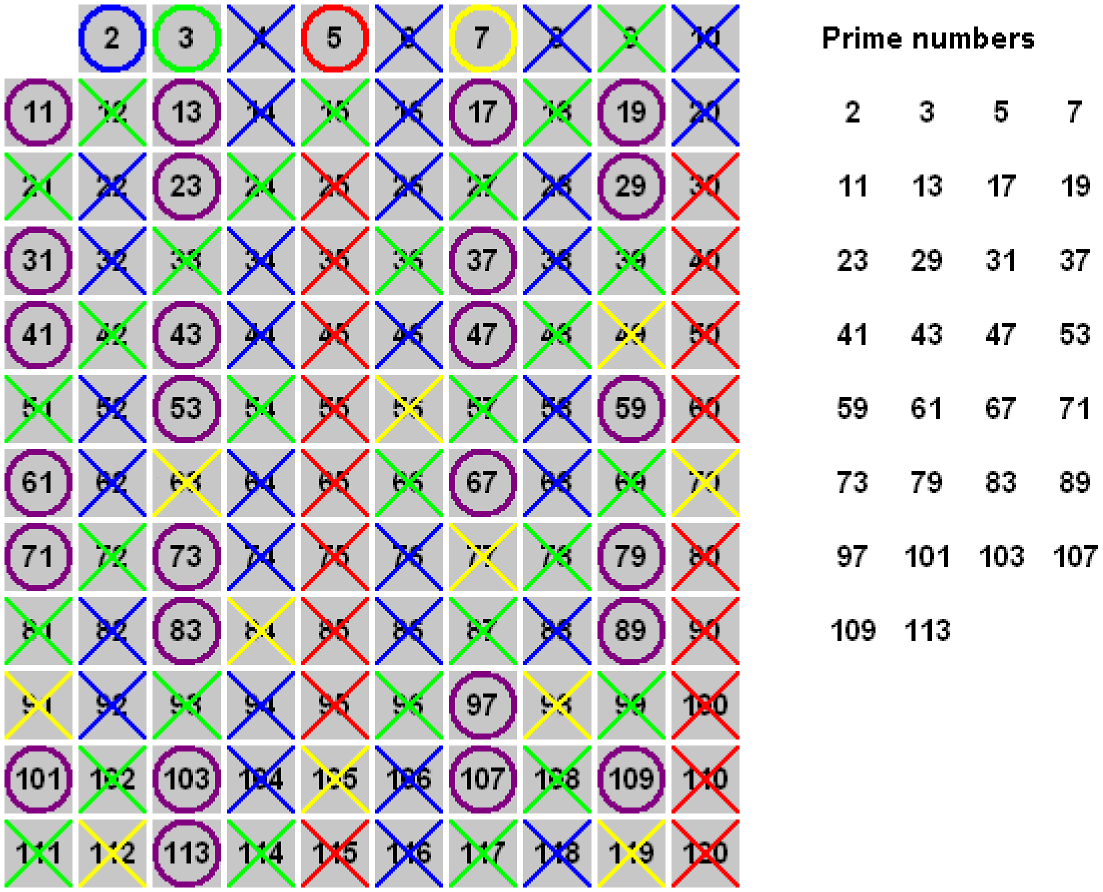
```python
def slow_list_primes(n):
    primes = []                      # empty list
    for suspect in range(2, n + 1):
        is_prime = True
        for prime in primes:
            if suspect % prime == 0:
                is_prime = False
                break
        if is_prime:
            primes.append(suspect)   # add item in list
    return primes

print(slow_list_primes(10))
```

```
[2, 3, 5, 7]
```

Third exercise

The sieve of Eratosthenes is a ancient algorithm for finding all prime numbers up to a given limit n.



Prime numbers

2 3 5 7

11 13 17 19

23 29 31 37

41 43 47 53

59 61 67 71

73 79 83 89

97 101 103 107

109 113

The Eratosthenes' method is as follow:

1. Consider a list of consecutive integers from 2 through n: (2, 3, 4, ..., n).
   Initially suppose they are all primes
2. Let p the first prime number of the list (initially $p = 2$ the smallest prime number).
   Enumerate the multiples of p ($2p$, $3p$, $4p$, ...) until n and mark them as 'not prime'
3. Find the first number greater than p in the list that is not marked 'not prime'
   (which is the next prime number after p)
   and repeat from strep 2

When the algorithm terminates, the numbers remaining not marked in the list are all the primes below n

Define a function '''list_prime(n)''' who use the Erathosthenes' method to find all primes until n.
Use a list of boolean indexed by 0:n to mark numbers.

In [16]:

```
#Exercise
def list_primes(n):
    """
    List all prime number below n
    My solution : [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 5
9]
    """
    #TODO
print(list_primes(60))
```

```
None
```

Solution

In [17]:

```
def list_primes(n):
    """
    List all prime number below n
    """

    # initialize result array
    primes = []

    # initialisation of the sieve using list concatenation (+) and repeat (*n)
    sieve = [False] * 2 + [True] * (n - 2)

    # iterate through the sieve
    for prime, is_prime in enumerate(sieve):
        if is_prime:

            # We find a prime! Save it
            primes.append(prime)

            # slicing to mark all mutliple as non prime
            sieve[::prime] = [False for i in range(0, n, prime)]
    return primes
print(list_primes(60))
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59]
```

Dictionnaries

They are usefull to store metadata

In [18]:

```
#define an empty dictonnary
my_emptydict = dict()
another_emptydict = {}
print(my_emptydict, another_emptydict)
print()

# define a dictionnary with items
personnal_information = {"lastname":"Gaston", "firstname":"Benoist"}
print(personnal_information)

# add new item
personnal_information["birthday"] = (10, 27)
print(personnal_information)
print(personnal_information["birthday"])
print()

# loop by key
for key in personnal_information:
    print(key)
print()

# loop by value
for value in personnal_information.values():
    print(value)
print()

# loop by items
for key, value in personnal_information.items():
    print(key, " : ", value)
```

```
{} {}

{'lastname': 'Gaston', 'firstname': 'Benoist'}
{'birthday': (10, 27), 'lastname': 'Gaston', 'firstname': 'Benois
t'}
(10, 27)

birthday
lastname
firstname

(10, 27)
Gaston
Benoist

birthday  :  (10, 27)
lastname  :  Gaston
firstname  :  Benoist
```

I/O

- Generally : use dedicated packages (pillow for images, ...) -> ecosystem

Strings

- String can be enclosed by single quotes ('...') or double ("..."). Use \ for escape quote

In [19]:

```
# single quote
print('Être ange')

# escape the second single quote
print('C\'est étrange')

# double quote enclosement single quote as a character
print("Dit l'ange")
```

```
Être ange
C'est étrange
Dit l'ange
```

- Strings can be concatenated by + operator and repeated by * operator

In [20]:

```
print(3*'Hip '+'Hourra')

a_long_string = ("This is "  # the + operator is optionnal here
                 "a long string ")
print(a_long_string)
```

```
Hip Hip Hip Hourra
This is a long string
```

- Mutilines strings

In [21]:

```
# with explicit special character
str1 = 'Être âne\nC\'est étrane\nDit l\'âne'

# with triple quotes
str2 = """Être âne
C'est étrane
Dit l'âne"""

if str1 == str2:
    print("The two strings are identical")
else:
    print("The two strings are different")
print()

print(str1)
print(str2)
```

```
The two strings are identical

Être âne
C'est étrane
Dit l'âne
Être âne
C'est étrane
Dit l'âne
```

- formatting strings
  - `print("un nombre : %d" % nombre)`
  - `print("un nombre : {:}".format(nombre))`
  - `print(f"un nombre : {nombre:}")` (only in Python 3.6)

In [22]:

```
print("{:<6} {:^5} | {:^5} | {:^5}".format("", "spam", "eggs", "bacon"))

format_array = "{:<6} {:^5.1f} | {:^5.1f} | {:^5.1f}"
print(format_array.format("David:", 0, 1, 5))
print(format_array.format("John:", 0, 2, 0))
print(format_array.format("Paul:", 1.5, 0, 0))
```

```
       spam  | eggs  | bacon
David:  0.0  |  1.0  |  5.0
John:   0.0  |  2.0  |  0.0
Paul:   1.5  |  0.0  |  0.0
```

Ascii files

Always decode in input
Always encode in output
open do it for you, but this is not always the case. Be careful

- read

In [23]:

```
# use of encoding strongly encouraged
# when working whith ascii files, utf-8 is used by default
for i, line in enumerate(open("README.md", encoding="utf-8")):
    print(line.split())
    if i>5:
        break
```

```
['#', 'Notebooks', 'pour', 'la', 'formation', 'Python']
[]
['Online', 'version', ':', '[https://github.com/pums974/formation_p
ython](https://github.com/pums974/formation_python)']
[]
['#', 'Python', 'under', 'Windows']
['Connect', 'with', 'your', 'credentials']
[]
```

- write

In [24]:

```
# using context
with open("output_filenamme", 'w', encoding="utf-8") as f:
    f.write("{:>10} is formated text".format("This"))
```

Forth exercise

A palindrome is a word (sequence of characters) which reads the same backward as forward, such as madam or radar.
Write the code who read lines in "data/filein.txt" and write in "fileout.txt" lines who are palindrom

In [25]:

```
for line in open("data/filein.txt", "r", encoding="utf-8"):
    print(line, end="")
# Exercise
#TODO
```

```
aba
baba
artre
palinnilap
```

Solution

In [26]:

```
with open("fileout.txt", "w", encoding="utf-8") as file_out:
    for line in open("data/filein.txt", "r", encoding="utf-8"):
        if line[:-1] == line[-2::-1]:                           #  Ignore the \n
 at the end of line
            file_out.write(line)
            print(line[:-1])
```

```
aba
palinnilap
```

You're almost ready to Pythonize!

(You'll also need modules)

Best practices

The zen of Python

In [27]:

```
import this # PEP 20
```

```
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 i
t.
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!
```

Respect PEP 8 (https://www.python.org/dev/peps/pep-0008/)

- 4 spaces no tabs
- do not reinvent the wheel
- wrong comments are worst than no comments, clear code is better than obscur code
- use docstrings

More on this

- The Hitchhiker’s Guide to Python! (http://docs.python-guide.org/en/latest/)
- http://www.scipy-lectures.org (http://www.scipy-lectures.org/)