Introduction to Python Programming

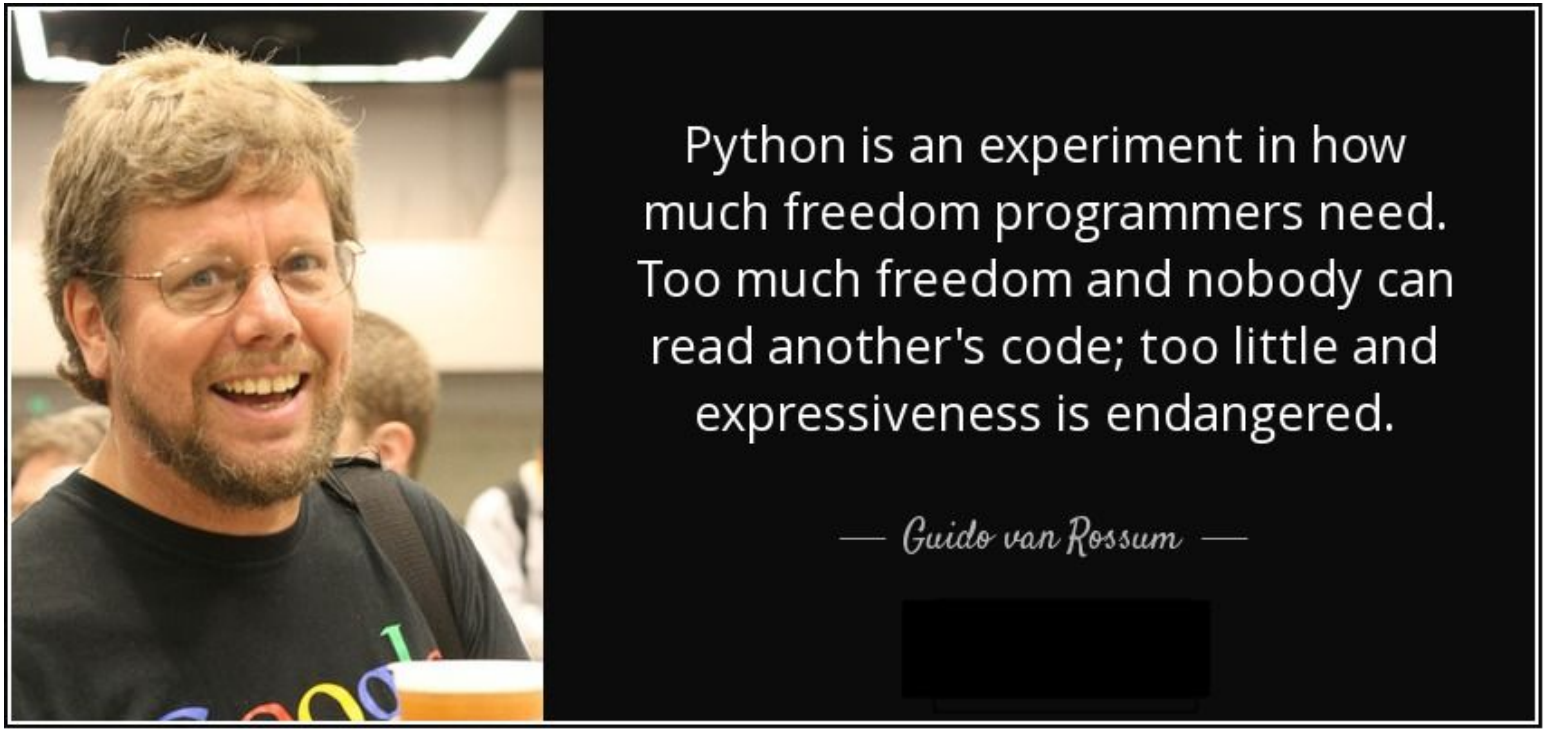
Marisol Flores, *III Summer School in Bioinformatics*

Outline

- Introduction
- Basic data types
- Basics for Python programming
- Some interesting libraries
- Exercises!!

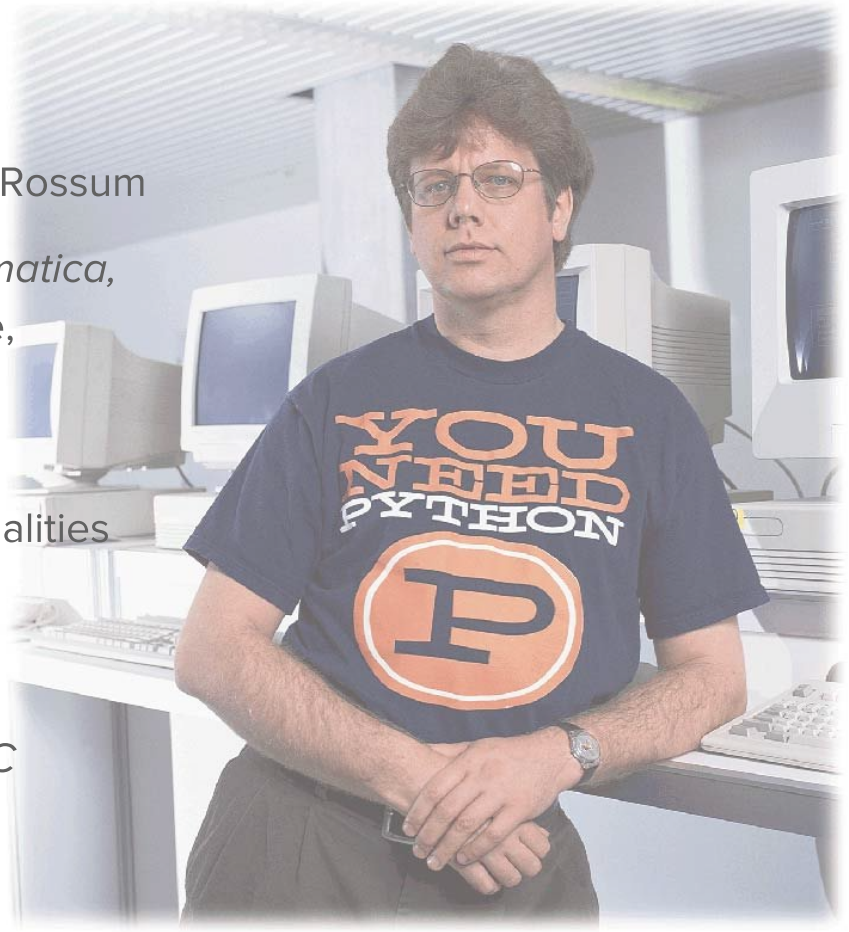
Introduction

The origins of Python



The origins of Python

- Idea and implementation: late 80s, Guido van Rossum
- Researchers team (*Centrum Wiskunde & Informatica,* Netherlands) developed ABC: general purpose, interactive, structured, high level, **intended for teaching or prototyping**
- Python: successor of ABC, with added functionalities and broader application range
- http://python-history.blogspot.com
- **Name:** *“something that would appeal to Unix/C hackers”*




Is there a life after death?


And here to discuss it
are three dead people.

Beautiful is better than ugly. **Explicit** is better than implicit. **Simple** is better than complex. **Complex** is better than complicated. **Flat** is better than nested. **Sparse** is better than dense. **Readability** counts. *Special cases* aren't special enough to break the rules.

Although **practicality** beats purity. *Errors* should never pass silently. Unless **explicitly** silenced. In the face of *ambiguity*, **refuse** the temptation to guess. There should be **one** — and preferably only one — obvious way to do it. Although that way may not be obvious at first *unless you're Dutch*. **Now** is better than never. Although never is **often** better than *right* now. If the implementation is *hard* to explain, it's a **bad** idea. If the implementation is *easy* to explain, it *may* be a **good** idea. **Namespaces** are one *honking great* idea — let's do more of those!

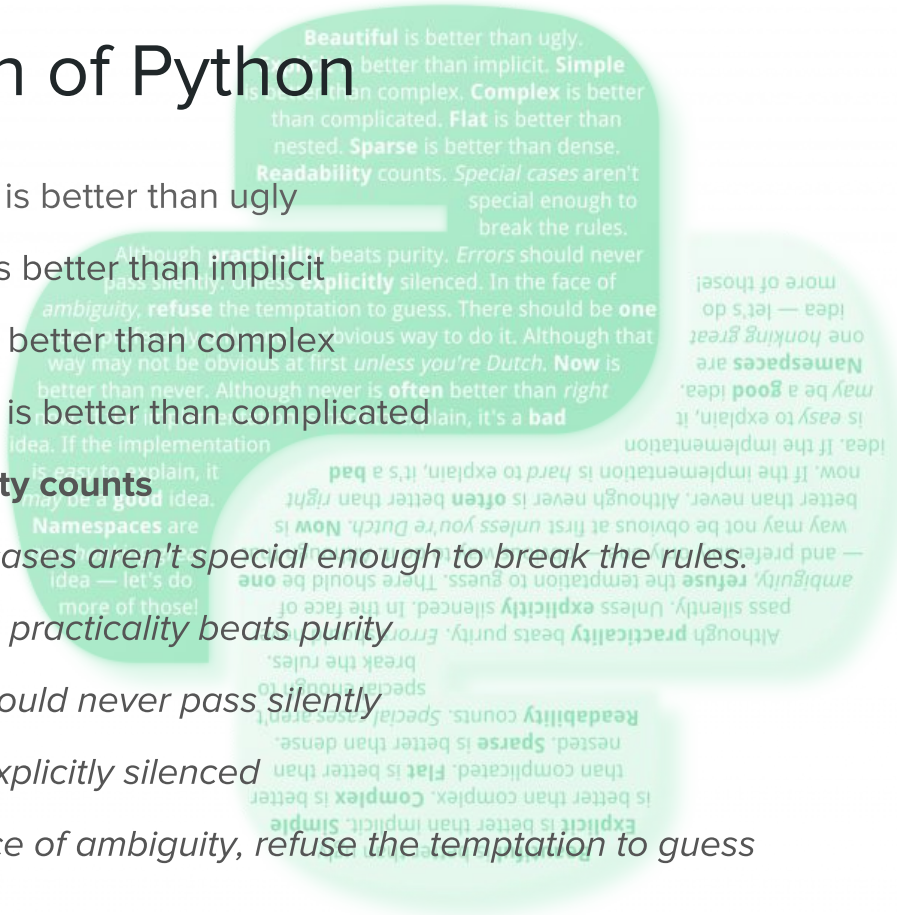
Beautiful is better than ugly. **Explicit** is better than implicit. **Simple** is better than complex. **Complex** is better than complicated. **Flat** is better than nested. **Sparse** is better than dense. **Readability** counts. *Special cases* aren't special enough to break the rules.

Although **practicality** beats purity. *Errors* should never pass silently. Unless **explicitly** silenced. In the face of *ambiguity*, **refuse** the temptation to guess. There should be **one** — and preferably only one — obvious way to do it. Although that way may not be obvious at first *unless you're Dutch*. **Now** is better than never. Although never is **often** better than *right* now. If the implementation is *hard* to explain, it's a **bad** idea. If the implementation is *easy* to explain, it *may* be a **good** idea. **Namespaces** are one *honking great* idea — let's do more of those!

python™

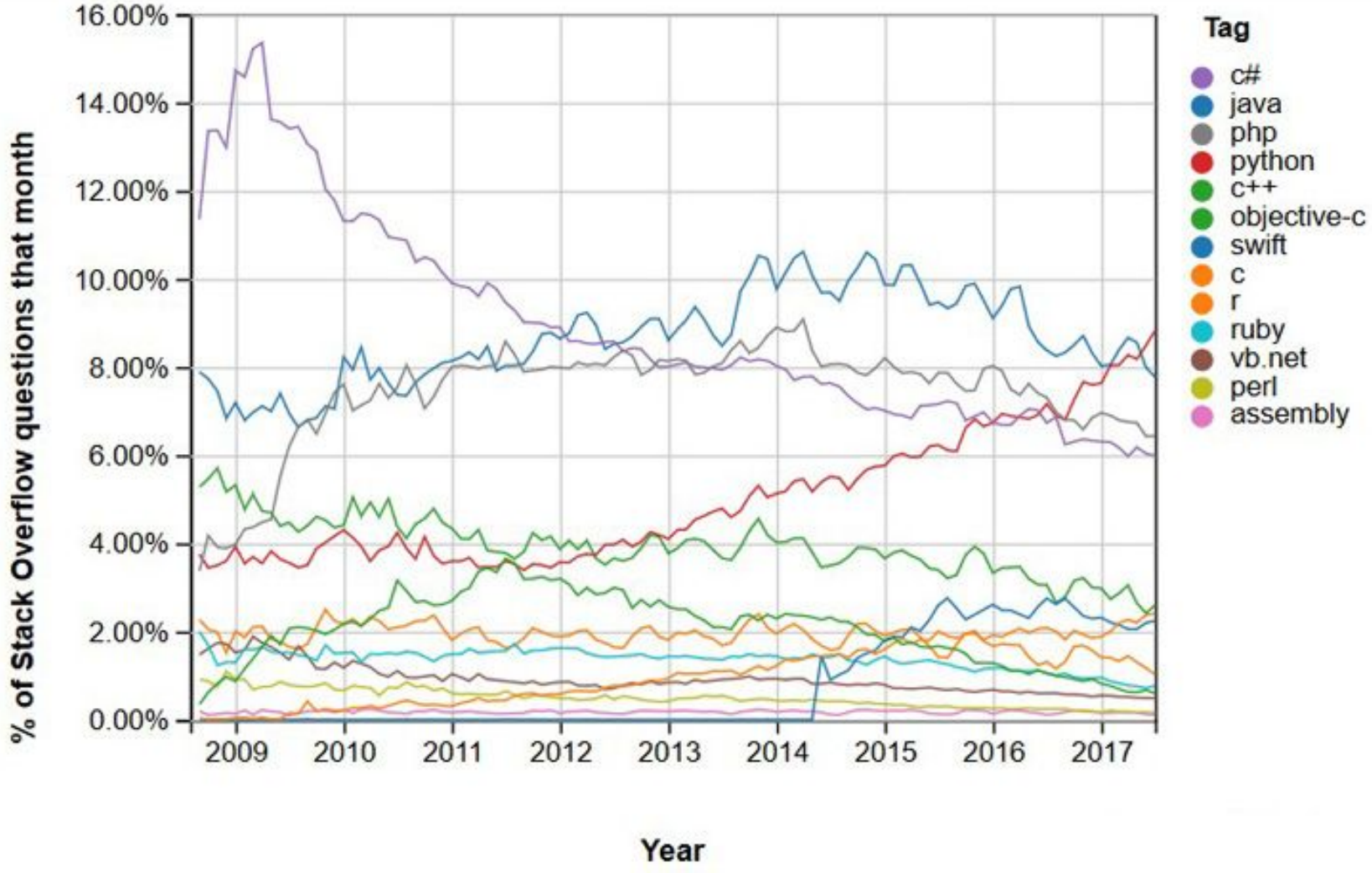
The Zen of Python

- Beautiful is better than ugly
- Explicit is better than implicit
- Simple is better than complex
- Complex is better than complicated
- **Readibility 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*



python

TM


% of Stack Overflow questions that month
16.00%
14.00%
12.00%
10.00%
8.00%
6.00%
4.00%
2.00%
0.00%
2009
2010
2011
2012
2013
2014
2015
2016
2017
Year
Tag
c#
java
php
python
c++
objective-c
swift
c
r
ruby
vb.net
perl
assembly

BLOG@CACM

Python is Now the Most Popular Introductory Teaching Language at Top U.S. Universities

By Philip Guo
July 7, 2014
Comments (10)

Summary

At the time of writing (July 2014), Python is currently the most popular language for teaching introductory computer science courses at top-ranked U.S. departments.

Specifically, eight of the top 10 CS departments (80%), and 27 of the top 39 (69%), teach Python in introductory CS0 or CS1 courses.

Readability

JAVA

```
public class Hello
{
    public static void main(String[] args) {
        System.out.printf("Hello world!");
    }
}
```

Python

```
print("Hello world!")
```

```c
#include <stdio.h>
int main(int argc, char **argv) {
  FILE *in, *out;
  int c;
  in = fopen("input.txt", "r");
  out = fopen("output.txt", "w");
  while ((c = fgetc(in)) != EOF) {
    fputc(c, out);
  }
  fclose(out);
  fclose(in);
}
```

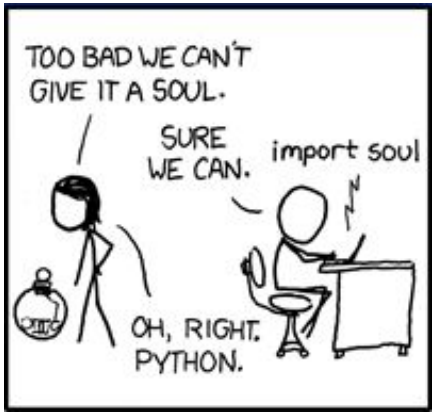
C

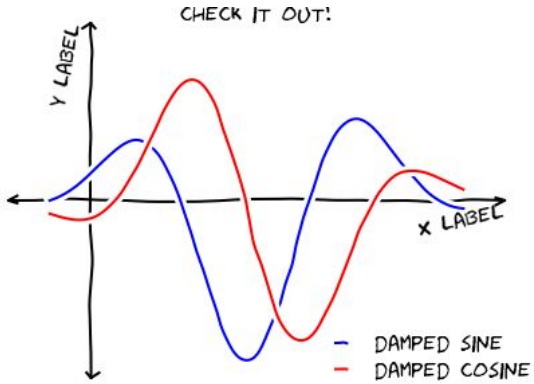
```python
in = open("input.txt")
out = open("output.txt", "w")
out.writelines(in)
in.close()
out.close()
```

Python

Also

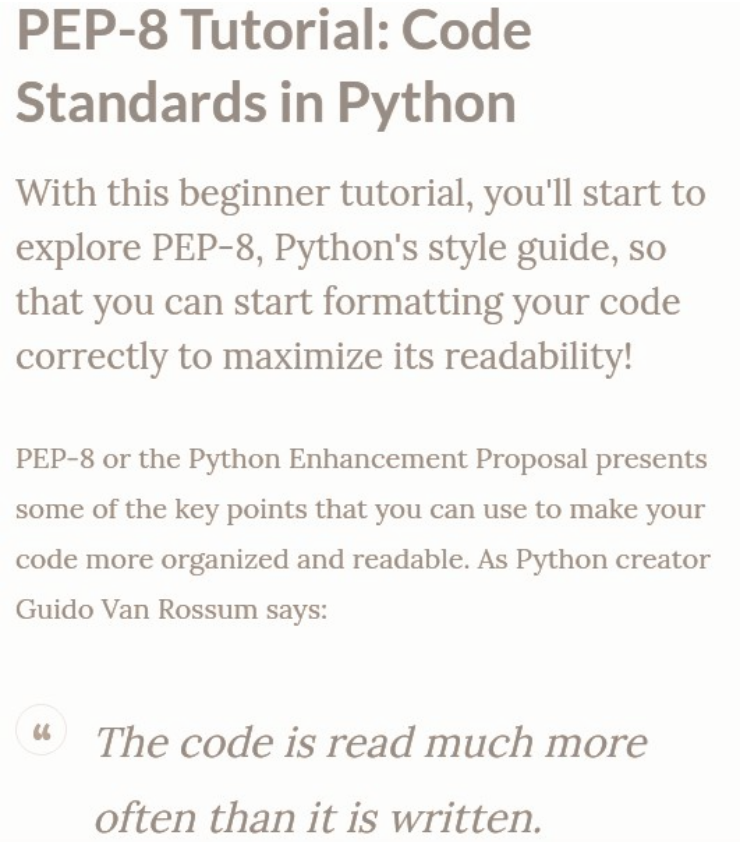
- Portability
- Multiparadigm
- LOTS of libraries FOR EVERYTHING
- Easy to use with other languages
- Open source
- Many users around the world (including Google, NASA, Pixas, BitTorrent,...)





Resources

- Tutorials
- Python Enhancement Proposal, PEP-8 (spaces, maximum line length, blank lines, ...)
- Books!



PEP-8 Tutorial: Code Standards in Python

With this beginner tutorial, you'll start to explore PEP-8, Python's style guide, so that you can start formatting your code correctly to maximize its readability!

PEP-8 or the Python Enhancement Proposal presents some of the key points that you can use to make your code more organized and readable. As Python creator Guido Van Rossum says:

> *The code is read much more often than it is written.*

Basics

First steps

- Install Python
- Install PIP
- **IPYTHON** is pretty good

Comments

```
# This is a comment
```

Simple calculations

Name	*Meaning*	*Example*	*Result*
`+`	Addition	`34 + 1`	`35`
`-`	Subtraction	`34.0 - 0.1`	`33.9`
`*`	Multiplication	`300 * 30`	`9000`
`/`	Float Division	`1 / 2`	`0.5`
`//`	Integer Division	`1 // 2`	`0`
`**`	Exponentiation	`4 ** 0.5`	`2.0`
`%`	Remainder	`20 % 3`	`2`

- Usual order of operations
- Floating-point results require at least one floating-point operator (Python 2)

[table from: thepythonguru.com]

Variables

- No need of declaring variables
- Case sensitive
- Letters, digits, underscore.
- Don't start with digits.
- Don't use keywords

Keywords

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

```
>> x = 7

>> y = 5.

>> z = x + y

>> a,b = 3,4

>> type(z)

>> float(x)
```

Strings

```
>> s1 = ‘Monday ’
>> s2 = “sleepy at UNAM early in the morning”
>> s3 = ‘this is a “valid” thing to do’
>> s4 = “this is NOT ok’
```

TRY:

```
>> s1 + s2
>> s1 * 3
>> s2.lower()
>> s2.upper()
>> s2.count(‘a’)
```

```
>> s2.replace(‘e’, ‘k’)
>> s2.split()
>> s2.split(‘n’)
>> ‘y’ in s2
>> len(s1)
```

Lists

```
>> a = [2,3,4]                          # collection of objects
>> b = [3,9,'nutella', 8]               # any object type
>> c = []                               # they could be empty
>> d = [1, [6,2], [10, [3,9]]]          # they could be nested
```

TRY:

```
>> a + b
>> b * 3
>> x = a[0]
>> a[-1]
>> y = b[1:3]
```

```
>> b[0] = -100
>> z = d[1][0]
>> len(b)
>> b.index(9)
>> b.reverse()
>> w = range(10)
```

Tuples

```
>> f = (2,3,‘abc’)
>> g = (5,)
>> h = (1, 2, (5,7))
```

TRY:

```
>> f[0]
>> f[0] = -100
```

Just like lists, but not **mutable**

Slices

```
>> S = ‘juriquilla’          # same thing for lists, strings, tuples
```

j	u	r	i	q	u	i	l	l	a
0	1	2	3	4	5	6	7	8	9
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

S[start: end] # end is NOT included

TRY:

```
>> S[3]
>> S[1:4]
>> S[1:]
>> S[:]
>> S[1:100]
```

```
>> S[-1]
>> S[-4]
>> S[:-3]
>> S[-3:]
>> S[-4:-2]
```

Dictionaries

```
>> biothing = {‘blood’:‘red’, ‘legs’:2, ‘bionumbers’: [1,2,3]}
```

TRY:

```
>> biothing[‘blood’]
>> len(biothing)
>> biothing.keys()
>> biothing.values()
>> biothing[‘legs’] = ‘maybe one’
>> biothing[‘hands’] = 5
>> biothing[3] = ‘whatever’
>> del biothing[‘blood’]
```

- Basic idea:

  D[*key*] = *value*

- Hash tables
- Only immutable objects can be keys
- Values could be anything, even dictionaries

4 ways to define a dictionary

1. One element at the time

```
>> D = {}   # empty dictionary
>> D['Italy'] = 'Rome'
```

2. List of tuples

```
>> L  = [('a',1), ('b',2)]
>> D2 = dict(L)
```

3. Combining two lists (same length)

```
>> A = [1,2,3]
>> B = ['rock', 'paper', 'lizard']
>> C = zip(A,B)
>> D1 = dict(C)
>> D2 = dict(zip(B,A))
```

4. Comprehension lists

Dictionaries

```
>> biothing = {‘blood’:‘red’, ‘legs’:2, ‘bionumbers’: [1,2,3]}
```

TRY:

```
>> biothing[‘blood’]
>> len(biothing)
>> biothing.keys()
>> biothing.values()
>> biothing[‘legs’] = ‘maybe one’
>> biothing[‘hands’] = 5
>> biothing[3] = ‘whatever’
>> del biothing[‘blood’]
```

- Basic idea:

  D[*key*] = *value*

- Hash tables
- Only immutable objects can be keys
- Values could be anything, even dictionaries

INPUT

```
>> N = raw_input(‘Your name? ’)        # Python 2
>> N = input(‘Your name? ’)            # Python 3
>> N
```

OUTPUT

```
>> print(N)
>> print(‘Hello world’)
>> print(‘Hello ’ + N)
>> q = 4
>> print(‘I have ’ + str(q) + ‘ questions’)
```

SCRIPTS

A bunch of python code in a text file with **.py** extension

amazingGuess.py

```
N = raw_input('Age? ')
n = int(N)

birthYear = 2018 - n

print("you were born in " + str(birthYear))
```

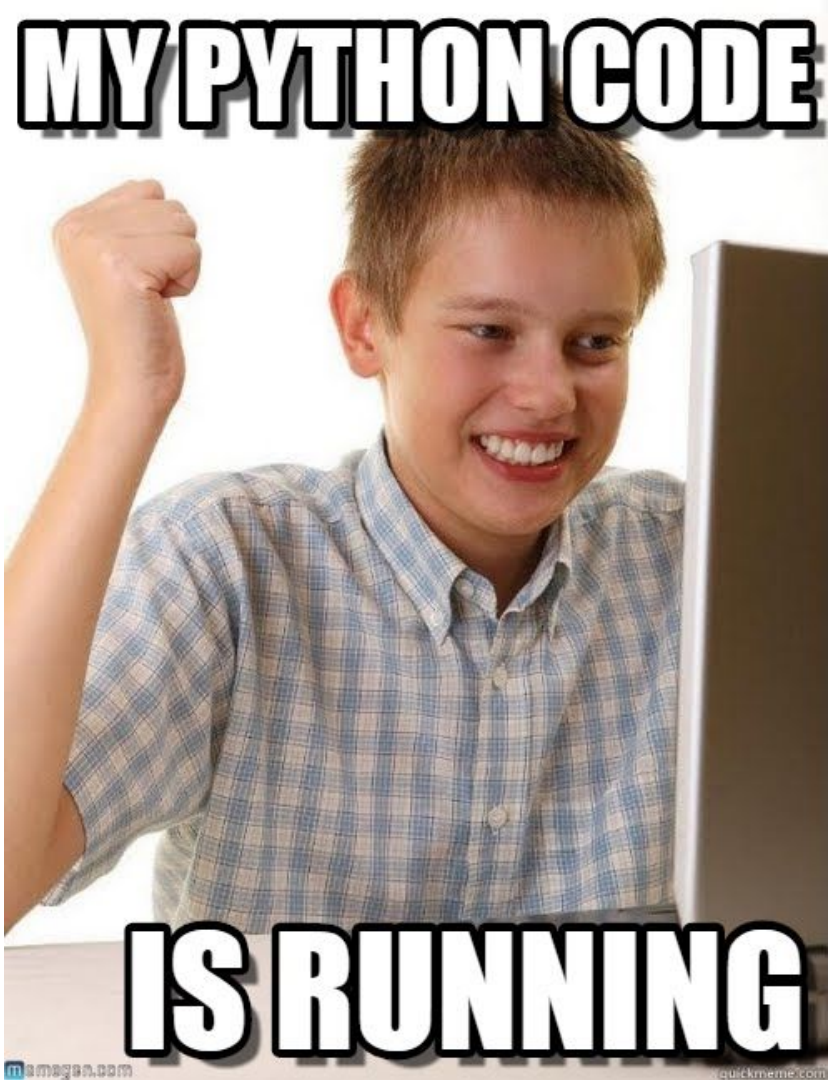
(Text editor)

```
# same directory than file

>> run amazingGuess
```

(Ipython terminal)

EXER CISES



Exercises

hardwork1.py	A script that asks for two numbers and prints the sum of them
hardwork2.py	A script that asks for your name and then YELLS AT YOU a personalized greeting (using your name)
hardwork3.py	Script that asks for your lunch’s bill and prints the amount to pay when a 10% tip is included

Programming in Python

Boolean operators

<, <=, >, >=, ==, !=, and, or

Conditional

```
if a <= b:
    if c != b:
        d = b + a
        e = d % 2
        print('Blah 1')
    else:
        print('abcde')
elif (a>b) and (c!=b):
    print(a)

print('Out of IF structure')
```

SPACES are really important

- They define blocks of code
- Beware of mixing tabs and spaces!
- Guideline: 4 spaces each level

While

```
b = 10
a = 0

while a < b:
    #something sciency

    a +=1
```

For

```
for i in [3,4,10,25]:
    print i

for letter in ‘juriquilla’:
    print letter

for k in range(25):
    print k
```

Exercise

Write a **script** that asks for a word and shows in the screen the score that word would get you in a scrabble game.

Functions

Python has **libraries** and **modules** with defined **functions**

```
>> import math
>> math.sqrt(5)

# or:
>> import math as M              # using alias
>> M.sqrt(5)

# or:
>> from math import sqrt         # import specific functions
>> sqrt(5)
```

Exercise

```
>> import random
>> L1 = range(50)
>> L2 = [‘hello’, ‘world’, ‘biology’, ‘stuff’]
```

TRY several times:

```
>> random.choice(L2)
>> random.sample(L1,n)                    #use different values for n
```

Our own functions

Define a function

```
def addTip(bill):
    total = 1.1 * bill
    return total
```

Save file as **restaurant.py**

We just defined a **module**, which can have several functions

Call function

```
>> import restaurant as res
>> res.addTip(147)
```

A module can have many functions

```
def addTip(bill):
    total = 1.1 * bill
    return total


def equallyDivide(bill,n=2):
    singleBill = float(bill)/n
    return singleBill
```

restaurant.py

Exercise

```
>> compareWeight(R1, R2)
Day1: Rat 1 is heavier than Rat 2
Day2: Both rats have the same weight
.
.
```

- Let’s say that you are doing experiments with rats in the lab. You have two lists, R1 and R2, specifying the daily weight of the rats. Write a function that compares both lists and prints the conclusion for every day.

Reading files

```
>> f = open('/home/marisol/path/testFile.txt', 'r')
>> textLines = f.readlines()
>> f.close()
```

Options	
'r'	reading
'r+'	reading and writing
'w'	writing
'a'	append lines
'rb', 'wb'	binary reading and writing

Methods to read files	
read( )	Single **string** with the **whole** text
readline( )	Single **string** with single line
readlines( )	**List of strings**, each string one line of the file

Writing files

```
>> f = open('/home/marisol/path/testFile.txt', 'w')
>> f.write('wow much biology such colors')
>> f.close()


>> f = open('/home/marisol/path/testFile.txt', 'a')
>> cheesyLines = ['turn around', 'every now and then',
                  'i get a little bit']
>> f.writelines(cheesyLines)
>> f.close()
```

Exercises

1. Using Python, create a new file **SUPERtotalEclipse.txt** with the same lines than **totalEclipse.txt**, but written in uppercase letters

   BONUS: replace the word `heart` with the word `moon`

2. Using Python, create a **function** that concatenates two given files with a specified name (your function, thus, requires three strings as input). Then use your function to create the file **karaoke.txt** with the contents of both **totalEclipse.txt** and **lifeInMars.txt**

Comprehension lists

```
>> L1 = [1,3,5,7,9,11,13]

>> L1 = [2*k+1 for k in range(7)]     # the pythonic way
```

```
L = [<expression> for var in <iterable>]

L = [<expression> for var in <iterable> if <condition>]
```

Exercise

```
>> S = “go go power rangers they know to use their
weapons only for defense”
```

- Create a list specifying the **length** of each word in **S**
- Create a list specifying the **length** of each word in **S,** but consider only the words that don’t have the letter **o**

Exercise

Write a **script** that asks for a word and shows in the screen the score that word would get you in a scrabble game.

Sorting a list

```
>> a = [5,1,4,3]
>> sorted(a)
>> s = [‘BBB’,’aaaa’,’CC’,’z’]
>> sorted(s)
>> sorted(s, reverse=True)
```

```
>> sorted(s, key=len)
```

Exercise

```
>> S = “go go power rangers they know to use their
weapons only for defense”
```

- Sort the words in **S** according to their **last** letter

  (you might have to define a function)

Warning: copying lists the right way

```
>> x = 3
>> y = x
>> y = 4
>> print(x)
    3
```

```
>> x = [3,7,10]
>> y = x
>> y[0] = -1000
>> print(x)
    [-1000, 7, 10]
```

How to copy lists?

```
>> x = y[:]
```

Useful libraries

Numpy

```
>> import numpy as np
>> M = numpy.array( [[10,20],[30,40]] )
>> 3*M
>> np.zeros([3,4])
>> np.linalg.qr(M)
>> b = np.array([[4],[5]])
>> x = np.linalg.solve(M,b)
>> np.dot(M,x)
```

NumPy for Matlab users

Introduction

MATLAB® and NumPy/SciPy have a lot in common. But there are many differences. NumPy and SciPy were created to do numerical and scientific computing in the most natural way with Python, not to be MATLAB® clones. This page is intended to be a place to collect wisdom about the differences, mostly for the purpose of helping proficient MATLAB® users become proficient NumPy and SciPy users.

Some Key Differences

In MATLAB®, the basic data type is a multidimensional array of double precision floating point numbers. Most expressions take such arrays and return such arrays. Operations on the 2-D instances of these arrays are designed to act more or less like matrix operations in linear algebra.	In NumPy the basic type is a multidimensional `array`. Operations on these arrays in all dimensionalities including 2D are elementwise operations. However, there is a special `matrix` type for doing linear algebra, which is just a subclass of the `array` class. Operations on matrix-class arrays are linear algebra operations.
MATLAB® uses 1 (one) based indexing. The initial element of a sequence is found using a(1). See note INDEXING	Python uses 0 (zero) based indexing. The initial element of a sequence is found using a[0].
MATLAB®'s scripting language was created for doing linear algebra. The syntax for basic	NumPy is based on Python, which was designed from the outset to be an excellent general-

`ode15s`	`scipy.integrate.ode(f).set_integrator('vode', method='bdf', order=5)`	integrate an ODE with BDF method

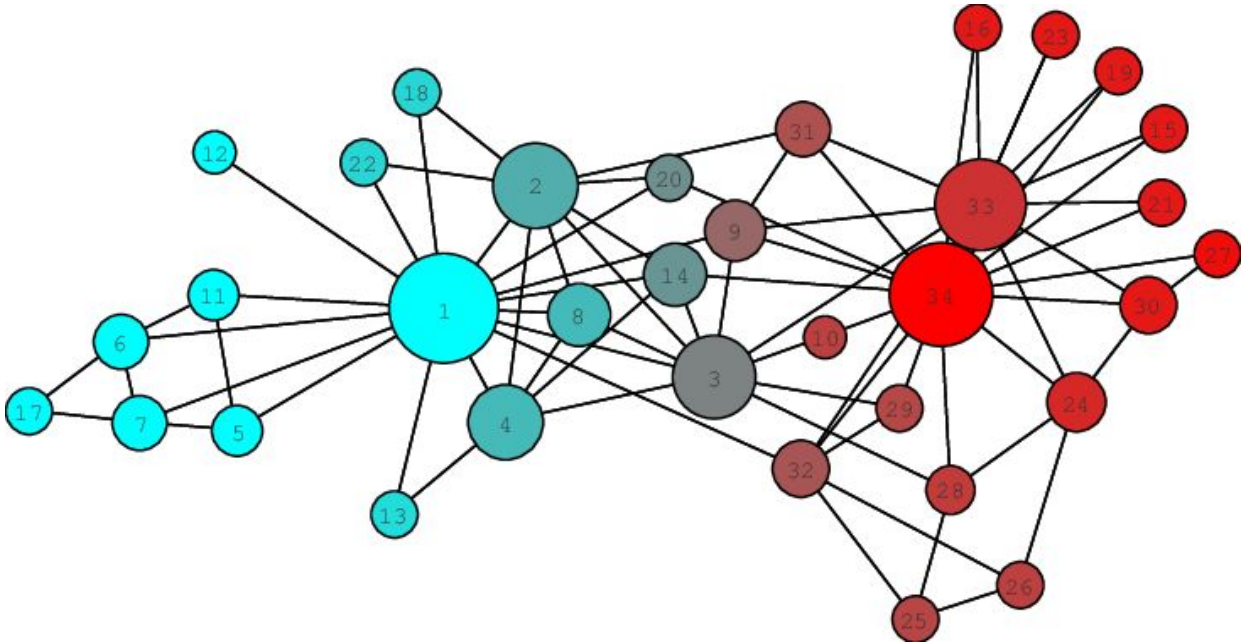
Linear Algebra Equivalents

MATLAB	NumPy	Notes
`ndims(a)`	`ndim(a)` or `a.ndim`	get the number of dimensions of an array
`numel(a)`	`size(a)` or `a.size`	get the number of elements of an array
`size(a)`	`shape(a)` or `a.shape`	get the "size" of the matrix
`size(a,n)`	`a.shape[n-1]`	get the number of elements of the n-th dimension of array `a`. (Note that MATLAB® uses 1 based indexing while Python uses 0 based indexing, See note INDEXING)
`[ 1 2 3; 4 5 6 ]`	`array([[1.,2.,3.], [4.,5.,6.]])`	2x3 matrix literal
`[ a b; c d ]`	`vstack([hstack([a,b]), hstack([c,d])])` or `bmat('a b; c d').A`	construct a matrix from blocks `a`, `b`, `c`, and `d`
`a(end)`	`a[-1]`	access last element in the 1xn matrix `a`
`a(2,5)`	`a[1,4]`	access element in second row, fifth column
`a(2,:)`	`a[1]` or `a[1,:]`	entire second row of `a`
`a(1:5,:)`	`a[0:5]` or `a[:5]` or `a[0:5,:]`	the first five rows of `a`
`a(end-4:end,:)`	`a[-5:]`	the last five rows of `a`
`a(1:3,5:9)`	`a[0:3][:,4:9]`	rows one to three and columns five to nine of `a`. This gives read-only access.
`a([2,4,5],[1,3])`	`a[ix_([1,3,4],[0,2])]`	rows 2,4 and 5 and columns 1 and 3. This allows the matrix to be modified, and doesn't require a regular slice.
`a(3:2:21,:)`	`a[ 2:21:2,:]`	every other row of `a`, starting with the third and going to the twenty-first
`a(1:2:end,:)`	`a[ ::2,:]`	every other row of `a`, starting with the first
`a(end:-1:1,:)` or `flipud(a)`	`a[ ::-1,:]`	`a` with rows in reverse order
`a([1:end 1],:)`	`a[r_[:len(a),0]]`	`a` with copy of the first row appended to the end
`a.'`	`a.transpose()` or `a.T`	transpose of `a`

Matplotlib

```
>> import matplotlib.pyplot as plt
>> x = numpy.array( [-2, -1, 0, 1, 2] )
>> y = x**2
>> plt.plot(x,y)
>> plt.show()
```

Networkx


16
23
19
18
31
15
12
22
2
20
33
21
9
27
14
34
11
1
8
30
6
10
3
17
7
5
4
29
24
32
28
13
26
25

Pandas