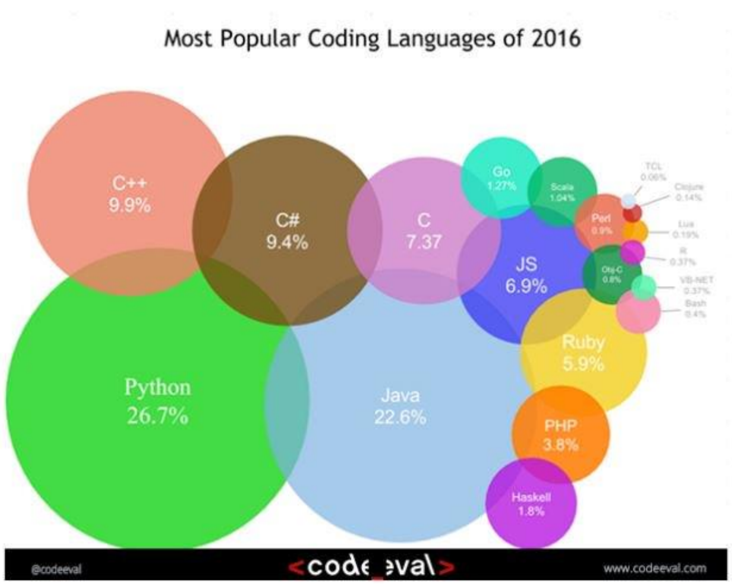
python™

Introduction to Python



https://www.python.org

UNIVERSITÉ TOULOUSE III PAUL SABATIER

python

- Simple
  - Python is a simple and minimalistic language in nature
  - Reading a **good** python program should be like reading English
  - Its Pseudo-code nature allows one to concentrate on the problem rather than the language
- Easy to Learn
- Free & Open source
  - Freely distributed and Open source
  - Maintained by the Python community
- High Level Language –memory management
- Portable

python

- Interpreted
  - You run the program straight from the source code.
  - Python program →Bytecode →a platforms native language
  - You can just copy over your code to another system and it will auto-magically work! *with python platform
- Object-Oriented
  - Simple and additionally supports procedural programming
- Extensible – easily import other code
- Embeddable –easily place your code in non-python programs
- Extensive libraries
  - (i.e. regular expressions, doc generation, CGI, ftp, web browsers, ZIP, WAV, cryptography, etc...) (wxPython, Twisted, Python Imaging library)

python Timeline/History

- Python was conceived in the late 1980s.
  - By Guido van Rossum
- In 1991 python 0.9.0 was published and reached the masses through alt.sources
- In January of 1994 python 1.0 was released
  - Functional programming tools like lambda, map, filter, and reduce
  - comp.lang.python formed, greatly increasing python's userbase

python Timeline/History

- In 1995, python 1.2 was released.
- By version 1.4 python had several new features
  - Keyword arguments (similar to those of common lisp)
  - Built-in support for complex numbers
  - Basic form of data-hiding through name mangling (easily bypassed however)
- Computer Programming for Everybody (CP4E) initiative
  - Make programming accessible to more people, with basic "literacy" similar to those required for English and math skills for some jobs.
  - Project was funded by DARPA

python Timeline/History

- In 2000, Python 2.0 was released.
  - Introduced list comprehensions
  - Introduced garbage collection

- In 2001, Python 2.2 was released.
  - Included unification of types and classes into one hierarchy, making pythons object model purely Object-oriented
  - Generators were added (function-like iterator behavior)

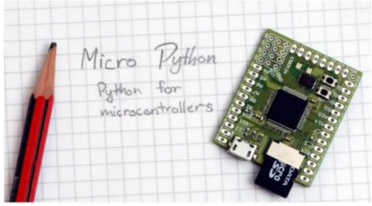
python Timeline/History

- In 2008, Python 3.0 (a.k.a. "Python 3000" or "Py3k") was released
  - Incompatible with 2.x versions
- Actually (March, 2019): Python 3.7.3
- Standards can be found on PEPs, Python Enhancement Proposals (http://legacy.python.org/dev/peps)

Python is used …

- In many projects such as:
  - blender

  - Aldebaran Robotics



  - micro Python for microcontrollers
  - SciPy.org, open-source software for mathematics and engineering

Presentation Overview

- Running Python and Output
- Data Types
- Input and File I/O
- Control Flow
- Functions
- Then, Why Python in Scientific Computation?
- Binary distributions Scientific Python

Hello World

- Open a terminal window and type “python”
- If on Windows open a Python IDE like IDLE
- At the prompt type ‘hello world!’

```
>>> 'hello world!'
'hello world!'
```

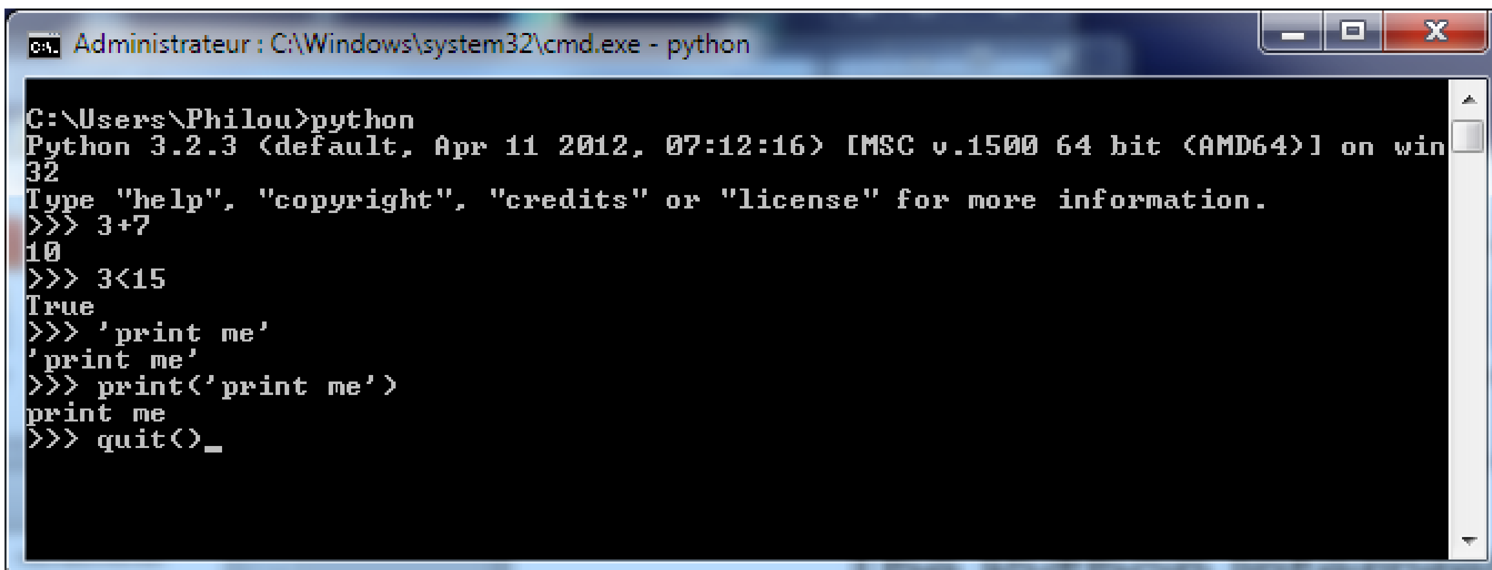
Python Overview

From *Learning Python, 2nd Edition*:

- Programs are composed of modules
- Modules contain statements
- Statements contain expressions
- Expressions create and process objects

The Python Interpreter

- Python is an interpreted language
- The interpreter provides an interactive environment to play with the language
- Results of expressions are printed on the screen



The print Statement

- Elements separated by commas print with a space between them

- A comma at the end of the statement (print('hello'),) will not print a newline character

```
>>> print 'hello'
Hello


>>> print('hello', 'there')
hello there
```

Documentation

The ‘#’ starts a line comment

```
>>> 'this will print'
'this will print'
>>> #'this will not'
>>>
```

Variables

- Are not declared, just assigned
- The variable is created the first time you assign it a value
- Are references to objects
- Type information is with the object, not the reference
- Everything in Python is an object

Everything is an object

- Everything means everything, including <u>functions</u> and <u>classes</u> (more on this later!)
- <u>Data type</u> is a property of the object and not of the variable

```
>>> x = 7
>>> x
7
>>> x = 'hello'
>>> x
'hello'
>>>
```

Numbers: Integers

- Integer – the equivalent of a C *long*
- Long Integer – an unbounded integer value.

```
>>> 132224
132224
>>> 132323 ** 2
17509376329L
>>>
```

Numbers: Floating Point

- int(x) converts x to an integer
- float(x) converts x to a floating point
- The interpreter shows a lot of digits

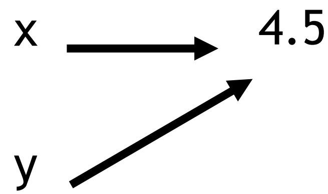
```
>>> 1.23232
1.2323200000000001
>>> print 1.23232
1.23232
>>> 1.3E7
13000000.0
>>> int(2.0)
2
>>> float(2)
2.0
```

Numbers: Complex

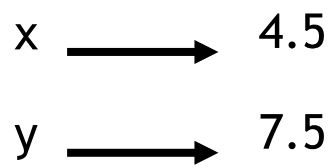
- Built into Python
- Same operations are supported as integer and float

```
>>> x = 3 + 2j
>>> y = -1j
>>> x + y
(3+1j)
>>> x * y
(2-3j)
```

Numbers are *immutable*




```
>>> x = 4.5
>>> y = x
>>> y += 3
>>> x
4.5
>>> y
7.5
```



String Literals

- Strings are *immutable*
- There is no char type like in C++ or Java
- + is overloaded to do concatenation

```
>>> x = 'hello'
>>> x = x + ' there'
>>> x
'hello there'
```

String Literals: Many Kinds

- Can use single or double quotes, and three double quotes for a multi-line string

```
>>> 'I am a string'
'I am a string'
>>> "So am I!"
'So am I!'
>>> s = """And me too!
though I am much longer
than the others :)"""
'And me too!\nthough I am much longer\nthan the others
:) '
>>> print s
And me too!
though I am much longer than the others :)'
```

Substrings and Methods

```
>>> s = '012345'
>>> s[3]
'3'
>>> s[1:4]
'123'
>>> s[2:]
'2345'
>>> s[:4]
'0123'
>>> s[-2]
'4'
```

- len(String) – returns the number of characters in the String
- str(Object) – returns a String representation of the Object

```
>>> len(x)
6
>>>
str(10.3)
'10.3'
```

String Formatting

- Similar to C's printf
- <formatted string> % <elements to insert>
- Can usually just use %s for everything, it will convert the object to its String representation.

```
>>> "One, %d, three" % 2
'One, 2, three'
>>> "%d, two, %s" % (1,3)
'1, two, 3'
>>> "%s two %s" % (1, 'three')
'1 two three'
>>>
```

Lists

- Ordered collection of data
- Data can be of different types
- Lists are *mutable*
- Issues with shared references and mutability
- Same subset operations as Strings

```
>>> x = [1,'hello', (3 + 2j)]
>>> x
[1, 'hello', (3+2j)]
>>> x[2]
(3+2j)
>>> x[0:2]
[1, 'hello']
```

Lists: Modifying Content

- **x[i] = a** reassigns the i^{th} element to the value a
- Since x and y point to the same list object, *both* are changed
- The method **append** also modifies the list

```
>>> x = [1,2,3]
>>> y = x
>>> x[1] = 15
>>> x
[1, 15, 3]
>>> y
[1, 15, 3]
>>> x.append(12)
>>> y
[1, 15, 3, 12]
```

Lists: Modifying Contents

- The method **append** modifies the list and returns **None**
- List addition (**+**) returns a new list

```
>>> x = [1,2,3]
>>> y = x
>>> z = x.append(12)
>>> z == None
True
>>> y
[1, 2, 3, 12]
>>> x = x + [9,10]
>>> x
[1, 2, 3, 12, 9, 10]
>>> y
[1, 2, 3, 12]
>>>
```

Tuples

- Tuples are *immutable* versions of lists
- One strange point is the format to make a tuple with one element:

  ',' is needed to differentiate from the mathematical expression (2)

```
>>> x = (1,2,3)
>>> x[1:]
(2, 3)
>>> y = (2,)
>>> y
(2,)
>>>
```

Dictionaries

- A set of key-value pairs
- Dictionaries are *mutable*

```
>>> d = {1 : 'hello', 'two' : 42, 'blah' :
[1,2,3]}
>>> d
{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}
>>> d['blah']
[1, 2, 3]
```

Dictionaries: Add/Modify

- Entries can be changed by assigning to that entry

```
>>> d
{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}
>>> d['two'] = 99
>>> d
{1: 'hello', 'two': 99, 'blah': [1, 2, 3]}
```

- Assigning to a key that does not exist adds an entry

```
>>> d[7] = 'new entry'
>>> d
{1: 'hello', 7: 'new entry', 'two': 99, 'blah': [1, 2, 3]}
```

Dictionaries: Deleting Elements

- The **del** method deletes an element from a dictionary

```
>>> d
{1: 'hello', 2: 'there', 10: 'world'}
>>> del(d[2])
>>> d
{1: 'hello', 10: 'world'}
```

Copying Dictionaries and Lists

- The built-in **list** function will copy a list
- The dictionary has a method called **copy**

```
>>> l1 = [1]
>>> l2 = list(l1)
>>> l1[0] = 22
>>> l1
[22]
>>> l2
[1]
```

```
>>> d = {1 : 10}
>>> d2 = d.copy()
>>> d[1] = 22
>>> d
{1: 22}
>>> d2
{1: 10}
```

Data Type Summary

- Lists, Tuples, and Dictionaries can store any type (including other lists, tuples, and dictionaries!)
- Only lists and dictionaries are mutable
- All variables are references

Data Type Summary

- **Integers**: 2323, 3234L
- **Floating Point**: 32.3, 3.1E2
- **Complex**: 3 + 2j, 1j
- **Lists**: l = [1,2,3]
- **Tuples**: t = (1,2,3)
- **Dictionaries**: d = {'hello' : 'there', 2 : 15}

Input

- The **input**(string) method returns a line of user input as a string
- The parameter is used as a prompt
- The string can be converted by using the conversion methods **int**(string), **float**(string), etc.

Input: Example

```
print ("What's your name?")
name = input("> ")

print("What year were you born?")
birthyear = int(input("> "))

print("Hi %s! You are %d years old!" % (name, 2019 - birthyear))
```

```
~: python input.py
What's your name?
> Philippe
What year were you born?
> 1969
Hi Philippe! You are 50 years old!
```

Files: Input

`inflobj = open('data', 'r')`	Open the file 'data' for input
`S = inflobj.read()`	Read whole file into one String
`S = inflobj.read(N)`	Reads N bytes (N >= 1)
`L = inflobj.readlines()`	Returns a list of line strings

UNIVERSITÉ TOULOUSE III PAUL SABATIER

Files: Output

`outflobj = open('data', 'w')`	Open the file 'data' for writing
`outflobj.write(S)`	Writes the string S to file
`outflobj.writelines(L)`	Writes each of the strings in list L to file
`outflobj.close()`	Closes the file

Booleans

- 0 and None are false
- Everything else is true
- True and False are aliases for 1 and 0 respectively

Boolean Expressions

- Compound boolean expressions short circuit
- and and or return one of the elements in the expression
- Note that when None is returned the interpreter does not print anything

```
>>> True and False
False
>>> False or True
True
>>> 7 and 14
14
>>> None and 2
>>> None or 2
2
```

Moving to Files

- The interpreter is a good place to try out some code, but what you type is not reusable
- Python code files can be read into the interpreter using the **import** statement

Moving to Files

- In order to be able to find a module called myscripts.py, the interpreter scans the list sys.path of directory names.
- The module must be in one of those directories.

```
>>> import sys
>>> sys.path
[‘’', 'C:\\WINDOWS\\system32\\python32.zip',
'C:\\Python32\\DLLs', 'C:\\Python32\\lib',
'C:\\Python32', 'C:\\Python32\\lib\\site-packages']
>>> import myscripts
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    import myscripts.py
ImportError: No module named myscripts.py
```

No Braces

- Python uses ***indentation*** instead of braces to determine the scope of expressions
- All lines must be indented the same amount to be part of the scope (or indented more if part of an inner scope)
- This **forces** the programmer to use proper indentation since the indenting is part of the program!

if Statements

```
import math
x = 30
if x <= 15 :
    y = x + 15
elif x <= 30  :
    y = x + 30
else :
    y = x
print ('y = '),
print (math.sin(y))
```

In file ifstatement.py

```
>>> import
ifstatement
y = -
0,3048106211022167
>>>
```

In interpreter

while Loops

```
x = 1
while x < 10 :
    print(x)
    x = x + 1
```

In whileloop.py

```
>>> import whileloop
1
2
3
4
5
6
7
8
9
>>>
```

In interpreter

Loop Control Statements

break	Jumps out of the closest enclosing loop
continue	Jumps to the top of the closest enclosing loop
pass	Does nothing, empty statement placeholder

The Loop else Clause

- The optional **else** clause runs only if the loop exits normally (not by break)

```
x = 1

while x < 3 :
    print(x)
    x = x + 1
else:

print('hello'
)
```

In whileelse.py

```
~: python whileelse.py
1
2
hello
```

Run from the command line

The Loop else Clause

```
x = 1
while x < 5 :
    print x
    x = x + 1
    break
else :
    print 'i got here'
```

whileelse2.py

```
~: python whileelse2.py
1
```

for Loops

- Similar to perl for loops, iterating through a list of values

forloop1.py

```
for x in
[1,7,13,2] :
    print(x)
```

```
~: python
forloop1.py
1
7
13
2
```

forloop2.py

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

```
~: python
forloop2.py
0
1
2
3
4
```

```
range(N) generates a list of numbers [0,1, …,
n-1]
```

for Loops

- **For** loops also may have the optional **else** clause

```
for x in range(5):
    print (x)
    break
else :
    print('i got here')
```

```
~: python elseforloop.py
1
```

elseforloop.py

Function Basics

```
def max(x,y) :
    if x < y :
        return x
    else :
        return y
```

```
>>> import functionbasics
>>> max(3,5)
5
>>> max('hello', 'there')
'there'
>>> max(3, 'hello')
'hello'
```

functionbasics.py

Functions are first class objects

- Can be assigned to a variable
- Can be passed as a parameter
- Can be returned from a function
- Functions are treated like any other variable in Python, the **def** statement simply assigns a function to a variable

Function names are like any variable

- Functions are objects
- The same reference rules hold for them as for other objects

```
>>> x = 10
>>> x
10
>>> def x () :
...     print('hello')
>>> x
<function x at 0x619f0>
>>> x()
hello
>>> x = 'blah'
>>> x
'blah'
```

Functions as Parameters

```
def foo(f, a) :
    return f(a)

def bar(x) :
    return x * x
```

```
>>> from funcasparam import *
>>> foo(bar, 3)
9
```

funcasparam.py

Note that the function foo takes two parameters and applies the first as a function with the second as its parameter

Higher-Order Functions

map(func,seq) – for all i, applies func(seq[i]) and returns the corresponding sequence of the calculated results.

```
def double(x):
    return 2*x
```

highorder.py

```
>>> from highorder import *
>>> lst = range(10)
>>> lst
[0,1,2,3,4,5,6,7,8,9]
>>> map(double,lst)
[0,2,4,6,8,10,12,14,16,18]
```

Higher-Order Functions

filter(boolfunc,seq) – returns a sequence containing all those items in seq for which boolfunc is True.

```
def even(x):
    return ((x%2 ==
0)
```

highorder.py

```
>>> from highorder import *
>>> lst = range(10)
>>> lst
[0,1,2,3,4,5,6,7,8,9]
>>> filter(even,lst)
[0,2,4,6,8]
```

Higher-Order Functions

reduce(func,seq) – applies func to the items of seq, from left to right, two-at-time, to reduce the seq to a single value.

```
def plus(x,y):
    return (x + y)
```

highorder.py

```
>>> from highorder import *
>>> lst = ['h','e','l','l','o']
>>> reduce(plus,lst)
'hello'
```

Functions Inside Functions

- Since they are like any other object, you can have functions inside functions

```
def foo (x,y) :
    def bar (z) :
        return z * 2
    return bar(x) + y
```

funcinfunc.py

```
>>> from funcinfunc import *
>>> foo(2,3)
7
```

Functions Returning Functions

```
def foo (x) :
    def bar(y) :
        return x + y
    return bar
# main
f = foo(3)
print (f)
print (f(2))
```

```
~: python
funcreturnfunc.py
<function bar at 0x612b0>
5
```

funcreturnfunc.py

Parameters: Defaults

- Parameters can be assigned default values
- They are overridden if a parameter is given for them
- The type of the default doesn't limit the type of a parameter

```
>>> def foo(x = 3) :
...     print (x)
...
>>> foo()
3
>>> foo(10)
10
>>> foo('hello')
hello
```

Parameters: Named

- Call by name
- Any positional arguments must come before named ones in a call

```
>>> def foo (a,b,c) :
...     print (a, b, c)
...
>>> foo(c = 10, a = 2, b = 14)
2 14 10
>>> foo(3, c = 2, b = 19)
3 19 2
```

Anonymous Functions

- A lambda expression returns a function object
- The body can only be a simple expression, not complex statements

```
>>> f = lambda x,y : x + y
>>> f(2,3)
5
>>> lst = ['one', lambda x : x * x, 3]
>>> lst[1](4)
16
```

Modules

- The highest level structure of Python
- Each file with the py suffix is a module
- Each module has its own namespace

Modules: Imports

`import mymodule`	Brings all elements of mymodule in, but must refer to as mymodule.<elem>
`from mymodule import x`	Imports x from mymodule right into this namespace
`from mymodule import *`	Imports all elements of mymodule into this namespace

To avoid some troubles …

- Define a function « **main** » in your module

```
def main():
    # my code here

if __name__ == "__main__": # if main fct
    main() # call it
```

UNIVERSITÉ TOULOUSE III PAUL SABATIER