

Python Tutorial

Dr. Xudong Liu
Assistant Professor
School of Computing
University of North Florida

Monday, 8/19/2019

Thanks to the MIT Python Course and the Python 3.6 Tutorial

Why Python?

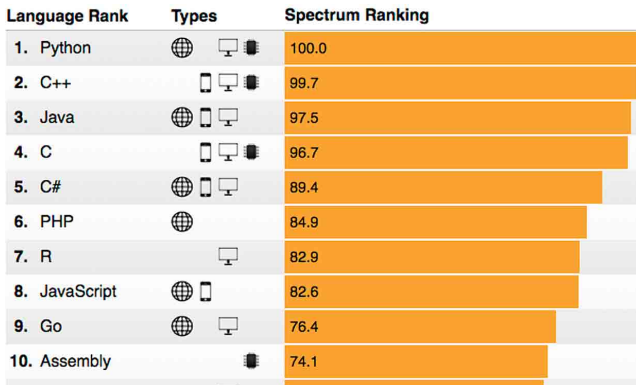


Figure: Top 10 Programming Languages by IEEE Spectrum 2018

Created by Dutch programmer Guido van Rossum in 1991, Python has long been one of the most popular programming languages among professionals.

Why Python?

- ① Python is arguably most popular in the general AI field, especially in machine learning.
- ② Python is easy to experiment with new ideas using minimal syntax.

Why “Python”?

About the origin, van Rossum wrote in 1996¹:

Over six years ago, in December 1989, I was looking for a “hobby” programming project that would keep me occupied during the week around Christmas. My office would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python’s Flying Circus).

¹https://en.wikipedia.org/wiki/Guido_van_Rossum

Python Basics

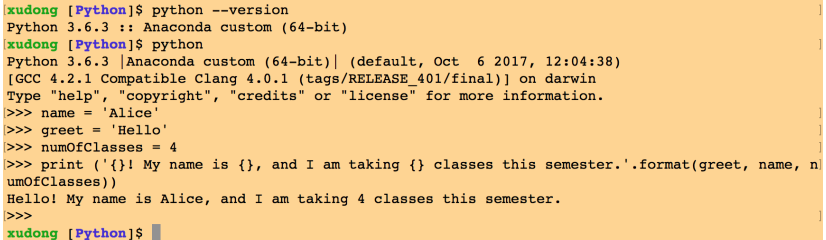
Python is a programming language that is

- ① very high level: high-level structures (e.g., lists, tuples, and dictionaries),
- ② **dynamic typing**: types of variables are inferred by the interpreter at runtime,
- ③ both compiled and interpreted: source code (.py) compiled to bytecode (.pyc) interpreted to machine code that runs, and
- ④ interactive: can be used interactively on command line.

The Python Interpreter

- 1 Invoke from the Terminal/cmd the Python interpreter by executing command: `python`
- 2 Make sure you are invoking Python 3.6. Check version by `python --version`
 - If your machine is installed multiple versions of Python, make sure you use version 3.6.
- 3 To exit from the interpreter, send a special signal called EOF to it: Control-D (Linux/Mac) or Control-Z (Windows).

The Python Interpreter



```
xudong [Python]$ python --version
Python 3.6.3 :: Anaconda custom (64-bit)
xudong [Python]$ python
Python 3.6.3 |Anaconda custom (64-bit)| (default, Oct 6 2017, 12:04:38)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> name = 'Alice'
>>> greet = 'Hello'
>>> numOfClasses = 4
>>> print ('{}\! My name is {}, and I am taking {} classes this semester.'.format(greet, name, numOfClasses))
Hello! My name is Alice, and I am taking 4 classes this semester.
>>>
xudong [Python]$
```

Figure: Screen shot of Python Interpreter

Run Python Scripts from the Command Line

- 1 Once you created a Python script (.py) using your favorite editor (e.g., Notepad, vi, emacs, and IDE), you can run it on the command line simply by `python example.py`
 - Again, make sure you run the right version of Python.

- 1 Download and install the latest version from <https://www.jetbrains.com/pycharm/>.
- 2 When creating a new project, choose python3.6 as the base interpreter.

Quick Python Tutorial

I assume you know well one of C, C++ and Java.

- ① Types
- ② Data structures
- ③ Control and looping statements
- ④ Functions and recursion
- ⑤ Modules
- ⑥ I/O
- ⑦ Class and Inheritance

Python Programs

- ① A Python program is a sequence of **statements** that are executed one after another by the Python interpreter.
- ② A statement could be as simple as an assignment (`taxRate=0.7`) and as complex as a definition of a function or a class.

Data Objects and Their Types

- 1 Python programs manipulate data **objects**.
 - In fact everything in Python is an object!²
- 2 An object has a **type** that defines what operations can be done on it.
 - `int`: addition, subtraction, multiplication, etc
 - `str`: concatenation, slicing, searching, etc
- 3 Generally, there are two kinds of types:
 - **Scalar** types: cannot be subdivided (e.g., `int` and `float`)
 - **Non-scalar** types: have internal structures (e.g., `function` and `classobj`)

²http://www.diveintopython.net/getting_to_know_python/everything_is_an_object.html (Read at your own risk)

Scalar Types

There are five scalar types in Python:

- ① `int`: the type of integer objects
- ② `float`: real numbers
- ③ `complex`: complex numbers (e.g., $1 + 2j$)
- ④ `bool`: Boolean values `True` and `False`
- ⑤ `NoneType`: special type and has only one value `None`
- ⑥ Can use built-in function `type` to tell the type of a data object:
`type(3.1415926)`.
- ⑦ Can cast types: `int(6.78)` gives 6 and `float(4)` gives 4.0.

Expressions

- ① Combine objects and operators to form **expressions**.
- ② Each expression is **evaluated** to a value of some type.
- ③ Operations on int and float:
 - $+$, $-$, $*$, $/$, $\%$, $**$
 - Result is int if both operands are int; float, otherwise.
- ④ complex expressions are evaluated considering **precedences** of operators.
- ⑤ Can use parentheses too.

Variables Binding Objects

- ❶ An **assignment** statement **binds** a data object or a variable to another variable.
 - `pi = 3.14`
 - `l = [1, 2, 3]`
 - `c = MyClass()`
 - `d = c`
- ❷ Assignment in Python is sort of **sneaky but important** to understand:
 - Data object: creates that object of some type and assigns a unique **identity** (like a reference or address) to the left handside. Can use built-in function `id` to see the identity of an object `id(pi)`.
 - Variable: simply assigns the identity of the right to the left.
- ❸ To retrieve values binded with variables, simply use the variable name.
- ❹ Re-binding: assign new objects to existing variables.
 - `l = {4, 5, 6, 7}`
 - Now `type(l)` should say `<type 'set'>`
 - Like Java, Python automatically collects garbage objects when no references to it.

Built-in Data Structures

- ① Strings
- ② Lists
- ③ Tuples
- ④ Sets
- ⑤ Dictionaries

Strings

- ① A **string** is a sequence of characters enclosed by quotations.
- ② Can compare using relational operators (`==`, `>=`, `>`, etc)
 - In Java you can't.
- ③ Built-in function `len` tells the length of a string.
- ④ Use brackets to access characters in a string. **Two** ways to index:
 - $0, 1, 2, \dots, n - 1$
 - $-n, -n + 1, -n + 2, \dots, -1$
- ⑤ Use built-in function `str` to cast other types to string: `str(1.234)`

Strings Quotations

Python provides four different types of quotations to enclose a string:

- ❶ Singles: `'Hello, world!'`
 - Have to escape single quotes: `'I don\'t know!'`
- ❷ Doubles: `"Hello, world!"`
 - Have to escape double quotes: `"\"Yes,\" she said."`
- ❸ Triple-singles: `'''Hello, world!'''`
 - Do not need to escape single quotes
- ❹ Triple-doubles: `"""Hello, world!"""`
 - Do not need to escape double quotes
 - Can make strings spanning multiple lines
 - By PEP-8 convention, we are suggested to use triple-doubles to make **docstrings**.

More on Strings Quotations

- 1 If a letter `r` is prepended to a string with whatever quotations, it defines a **raw** string that ignores all escaped codes.
 - `r"\\python\\n"` will be printed exactly those 10 characters.
- 2 Triple-single and triple-double quotes can make strings spanning multiple lines.
- 3 By PEP-8 convention, we are suggested to use triple-doubles to make **docstrings**.

Manipulating Strings

- ❶ String variables: `s1 = 'Hello, '`
- ❷ Concatenation: `s2 = s1 + 'world!'`
- ❸ Indexing: `print (s2[7]+s2[8]+s2[9]+s2[-2])`
- ❹ Slicing: `print (s2[1:5], s2[7:], s2[:])`
- ❺ Built-in methods³:
 - `s1=s1.replace('ll', 'LL')`: What has changed for `s1`?
 - `data=s2.split(" ")`

³<https://docs.python.org/2/library/string.html>

Mutability

- ① String objects are **immutable**, cannot be changed.
 - Similar to strings specified with `final` in Java or `const` in C/C++.

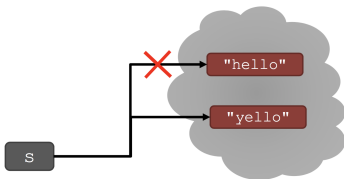
```
s = "hello"
```

```
s[0] = 'y'
```

```
s = 'y'+s[1:len(s)]
```

→ gives an error

→ is allowed,
s bound to new object



Lists

- ① A list is an **ordered**⁴ sequence of data, possibly of different types.
- ② A list can be written as a list of comma-separated objects between square brackets.
 - To create an empty list: `emptyList = []`
 - Can be heterogeneous: `l = [3, 3.14, 'pi']` (Not common)
 - Can be nested: `L = [['Allice', 'Bob', True], ['Alice', 'Joe', False]]`
- ③ As with strings, elements in a list can be accessed by their indices.
- ④ Lists are **mutable** so elements can be changed.
 - `L[0][0] = 'Alice'`

⁴in the sense that it **can be** ordered and elements can be indexed by their positions.

Operations on Lists

- ❶ Append elements to end of lists: `l.append(newElem)`
- ❷ Concatenate two lists using `+`
- ❸ Extend lists: `l.extend(anotherList)`
- ❹ Delete element at an index: `del l[2]`
- ❺ Remove element at end of list: `l.pop()`
- ❻ Search and remove element: `l.remove(targetElem)`
- ❼ Reverse a list: `l.reverse()`
- ❽ Sort a list:
 - `sorted(l)`: returns sorted list and doesn't mutate `l`.
 - `l.sort()`: mutates `l`.

Tuples

- ❶ Like a list, a **tuple** is an ordered sequence of elements, possibly of different types.
- ❷ A **tuple** is written as elements separated by commas with or without parentheses enclosing the values.
 - A good practice is to have parentheses.
 - `emptyTuple = ()`
 - `t = (2, 3, 4)`
 - `T = (('01202018', True), ('02042018', False))`
(Nested tuple)
- ❸ As with strings and lists, elements in a list can be accessed by their indices.
- ❹ Unlike lists, tuples are **immutable**.
 - So `t[0] += 1` gives an error.

Operations on Tuples

- 1 Get number of elements: `len(t)`
- 2 Concatenate two tuples using `+`
- 3 Sort a list: `sorted(t)`
 - `t.sort()`: can't.
- 4 Swap variable values: `(x, y) = (y, x)`
- 5 Unpacking: `x, y, z = t`

- ① A **set** is an unordered collection of **unique** elements, possibly of different types.
- ② A **set** is written as elements separated by commas with braces enclosing the values.
 - `emptySet = set(), {}` gives empty dictionary.
 - `s1 = {2, 3, 4}`
 - `s2 = {3, 45}`
 - Can't be nested.
- ③ Elements in a list cannot be accessed by indices.
 - Because unordered. But can be access iteratively.
- ④ Like lists, sets are **mutable**.

Operations on Sets

- 1 Get number of elements: `len(s1)`
- 2 Add elements: `s1.add(34)`
- 3 Remove elements: `s1.remove(2)`
- 4 Check membership: `4 in s1` gives `True`
- 5 Difference: `s1 -= s2`, `s3 = s1 - s2`
- 6 Intersection: `s3 = s1 & s2`
- 7 Union: `s4 = s1 | s2`

Dictionaries

- ❶ A **dictionary** is an unordered collection of **key-value** pairs
- ❷ A **dictionary** is written as *key:value* pairs separated by commas with braces enclosing these pairs.
 - Keys must be unique and **immutable**
 - So far immutables: int, float, complex, bool, str and tuple. (This is why there is no ++ or -- in Python.)
 - `emptyDict = {}.`
 - `tel = {'Alice': 4098, 'Bob': 4139}`
 - `print (tel['Bob'])`
 - `movieRatings = {'Alice':{'Batman':4.5}, 'Bob':{'Batman':2.5}}`
 - `print (movieRatings['Alice']['Batman'])`
- ❸ Elements in a dictionary cannot be accessed by indices.
 - Because unordered. But can be access iteratively.
- ❹ Dictionaries are **mutable**.

Operations on Dictionaries

- ① Get number of pairs: `len(tel)`
- ② Add elements: `tel['Phil']=3900`
- ③ Remove elements: `del tel['Bob']` or `name = tel.pop('Bob')`
- ④ Check membership: `'Alice' in tel` gives `True`
- ⑤ Get list of keys: `keysList = movieRatings.keys()`

- ① Types
- ② Data structures
- ③ Control and looping statements
- ④ Functions and recursion
- ⑤ Modules
- ⑥ I/O
- ⑦ Class and Inheritance

Comparison Operators

- ① Work directly on `int`, `float` and `str` types.
- ② Need extra effort to work on other types.
- ③ `>`, `>=`, `<`, `<=`, `==`, `!=`
 - Takes two operands and returns a `bool` value.
- ④ `==` vs. `is`
 - comparing data objects vs. comparing identities

Logical Operators

- ① `not`, `and`, `or`
 - Takes two `bool` operands and returns a `bool` value.

Control Flow

```
if <condition>:  
    <expression>  
    <expression>  
    ...
```

```
if <condition>:  
    <expression>  
    <expression>  
    ...  
else:  
    <expression>  
    <expression>  
    ...
```

```
if <condition>:  
    <expression>  
    <expression>  
    ...  
elif <condition>:  
    <expression>  
    <expression>  
    ...  
else:  
    <expression>  
    <expression>  
    ...
```

- 1 Conditions are evaluated to Truth or False.
- 2 Python uses **indentation** for blocks.
 - Recommended to use 4 spaces per indentation level.
 - Don't mix spaces with tabs.
- 3 Lines starting with # are comments.

Iterations Using while and for

```
while <condition>:    for <variable> in range(<some_num>):  
    <expression>        <expression>  
    <expression>        <expression>  
    ...                ...
```

- 1 Use `continue` and `break` to skip certain iterations.
- 2 Blocks after the colon are required.
 - The `pass` statement does nothing.
 - Put the `pass` statement there if no action is needed.
- 3 Built-in function `range(start, stop, step)` returns a list of integers from `start` until `stop` with `step` increment.

Iterating through Data Structures

```
s = 'Hello, world!'
numLetters = 0
for i in range(len(s)):
    if s[i].isalpha():
        numLetters += 1
```

```
s = 'Hello, world!'
numLetters = 0
for c in s:
    if c.isalpha():
        numLetters += 1
```

- 1 The one on the right is more Pythonic and preferred.
- 2 Iterating through lists, tuples, sets is similar.

Iterating through Dictionaries

```
d = {'Alice': 40, 'Bob': 39}
for key in d:
    print (key, 'is', d[key], 'years old.')
```

```
d = {'Alice': 40, 'Bob': 39}
for key, val in d.items():
    print (key, 'is', val, 'years old.')
```

Comprehensions

① List Comprehension:

- `[EXPRESSION for VAR in LIST if CONDITION]`
- `L = [e**2 for e in range(0,100) if e%10]`

② Set Comprehension:

- `{EXPRESSION for VAR in SET if CONDITION}`

③ Dictionary Comprehension:

- `{EXPRESSION for VAR in DICTIONARY if CONDITION}`

④ Tuple Comprehension:

- `tuple(EXPRESSION for VAR in TUPLE if CONDITION)`
- `S=1,2,3,4,5`
- `T = tuple(e*2 for e in S if e%2)`

Functions

```
def func(parameter1, parameter2):  
    """  
    Docstring  
    """  
    <code block>  
    return value
```

- 1 Parameters can have default values.
- 2 The return statement is optional. In case of absence, the function returns `None`.
- 3 Use `pass` for the block as placeholder for future implementation.

Default Arguments

- Default arguments in a function definition are specified as assignments. This creates a function that can be called with fewer arguments than it is defined to allow.

```
def ask(prompt, retries=4, complaint='Yes or no!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise IOError('refuse user')
        print (complaint)
```

```
ask("Do you want to go?")
```

Pass by Value or Reference?

- ❶ Pass by identity: when a function is called, the identity of the parameter variable is passed to function definition.

```
name = 'Adam'
names = []
names.append(name)

otherNames = names
otherNames.append('Bill')

print (names)
print (otherNames)
```


Pass by Value or Reference?

```
def foo(bar):  
    bar.append(100)  
    print (bar)
```

```
l = [10]  
foo(l)  
print (l)
```

```
def foo(bar):  
    bar = [100]  
    print (bar)
```

```
l = [10]  
foo(l)  
print (l)
```

```
def foo(bar):  
    bar = 'hello'  
    print (bar)
```

```
s = 'hi'  
foo(s)  
print (s)
```

- What are the prints?
- Note assignments and mutability.

- ❶ **Scopes** of variables dictate the life time of them.
 - Data objects, once created, live throughout until garbage collector recycles it.
 - Variables live from the moment it is created till the end of the scope they were created in.
- ❷ From inner to outer, we have the **local** scope, **enclosing-function** scope, the **global** scope and **built-in** scope.
- ❸ When trying to retrieve a variable to use, Python goes through these four scopes in that order (**LEGB**).
- ❹ Within a local or enclosing-function scope, if you want to write the global variable, you have to declare it in that scope with keyword **global**.

Scopes

```
# Global scope
i = 1

def f():
    print ('f(): ', i)

def g():
    i = 2
    print ('g(): ', i)

def h():
    global i
    i = 3
    print ('h(): ', i)
```

Scopes

```
# Global scope
print ('global : ', i)
f()
print ('global : ', i)
g()
print ('global : ', i)
h()    print ('global : ',
i)
```

Recursive Functions: Fibonacci Numbers

```
def Fib(n=10):  
    if n==0 or n==1:  
        return 1  
  
    return Fib(n-1)+Fib(n-2)
```

Recursion

```
def Fib(n=10):  
    a,b = 1,1  
    i = 2  
    while i<=n:  
        a,b = b,a+b  
        i += 1  
    return b
```

Iteration

- ① Types
- ② Data structures
- ③ Control and looping statements
- ④ Functions and recursion
- ⑤ Modules
- ⑥ I/O
- ⑦ Class and Inheritance

- 1 A **module** is a Python file (MYMODULE.py) that is a collection of definitions.
- 2 Definitions in a module can be **imported** into Python scripts:

- `import MYMODULE`
`MYMODULE.func()`
- `import MYMODULE as M`
`M.func()`
- `from MYMODULE import func, anotherFunc`
`func()`
- `from MYMODULE import *`
`func()`

Note this way, anything starting with a single underscore won't be imported.

- 1 Previously, we assumed your Python scripts and modules are in the same directory. What if they are not?
- 2 Python imports modules by searching the directories listed in `sys.path`.
 - ```
import sys
```
  - ```
print ('\n'.join(sys.path))
```
- 3 If you have your own modules in a directory say `DIR`, in a Python script before importing modules in it, you want to add `DIR` to `sys.path`:
 - ```
sys.path.append(DIR)
```
  - Note `DIR` must be the full path to the directory.



# File I/O

```
filename = raw_input("Enter file name: ")
try:
 f = open(filename, 'r')
except IOError:
 print ('cannot open', filename)
for line in f:
 print (line)
f.close()
```

## (a) Reading

```
f = open('tmp.txt', 'w')
f.write('Hello, class!\n')
f.close()
```

## (b) Writing

# Parsing CSV

```
import csv
try:
 f = open('example.csv', 'r')
except IOError:
 print ('cannot open it')
lines = csv.reader(f)
for line in lines:
 print (','.join(line))
f.close()
```

# Classes

```
class className:
 """
 Docstring
 """
 <statement-1>
 ...
```

- 1 A statement in a class could be a definition of **data attribute** or **function attribute**.
- 2 Data attributes can be shared among all class instances (**class var**) or specific to each class instance (**instance var**).
  - Variables defined in the class scope are shared among all instances.
  - Variables defined using the `self` keyword within the constructor `__init__` are specific to each instance.
  - No variables in a class can be **private**.
- 3 Function attributes can also be **class function** or **instance function**.

# Classes

```
class Person:
 numPersonObjs = 0

 def __init__(self, name, age):
 self.name = name
 self.age = age
 Person.numPersonObjs += 1

 def __str__(self):
 return self.name+": "+str(self.age)+" years old"

 def greet(self):
 print ("Hello, "+self.__str__())

 @classmethod
 def printNumPersonObjs(cls):
 print ("Num of persons: "+str(cls.numPersonObjs))
```

```
p1 = Person('Bill', 28)
print (p1)
p1.greet() # Equivalent to calling Person.greet(p)
p2 = Person('Kate', 43)
p3 = Person('Kurt', 32)
Person.printNumPersonObjs()
```

# Inheritance

- ① Child class inherits **all** attributes of the parent class.
- ② Child class can add more attributes.
- ③ Child class can override functions in the parent class.

# Inheritance

```
class Student (Person):
 numStudentObjs = 0

 def __init__(self, name, age, id):
 Person.__init__(self, name, age)
 self.id = id
 self.classmates = set()
 Student.numStudentObjs += 1

 def __str__(self):
 return Person.__str__(self)+' ('+str(self.id)+')'

 def addClassmate(self, classmate):
 self.classmates.add(classmate)

 @classmethod
 def printNumStudentObjs(cls):
 print ("Num of students: "+str(cls.numStudentObjs))
```

# Inheritance

```
p1 = Person('Bill', 28)
p2 = Person('Kate', 43)
p3 = Person('Kurt', 32)
s1 = Student('Alice', 22, 1234)
print (s1)
Person.printNumPersonObjs()
Student.printNumStudentObjs()
s1.addClassmate(p2)
s1.addClassmate(p3)
for e in s1.classmates:
 print (e)
s1.greet()
```