Web Programming
Python Part I.

Shuo Zhang

Python

- Extremely readable and versatile programming language

```
print(“hello python”)
```

- Multiparadigm language:
  - Scripting
  - Object-oriented
- One of most popular languages in data science

Python VS Java

- Python runs much slower in comparison to Java
- Python snippet takes very less memory in comparison to Java
- Python snippet is 3-5 times shorter than a Java snippet for the same functionality.
- Java is categorized as low-level implementation language, on the other hand, Python is high-level or glue type language.

Python VS C++

- Python snippet is comparatively 5-10 times shorter than a C++ snippet for the same functionality.
- Python is a high-level language and C++ is low level.
- Python acts as a glue language that used to combine components written in C++.
- Python provides much flexibility in calling functions and returning values in comparison to C++.
- C++ snippets works faster than Python.
- Python is interpreted while C++ is a pre-compiled.
- Python uses Garbage Collection whereas C++ doesn’t.

Python 2 VS Python 3

- They should not be thought of as entirely interchangeable
- Differences in code syntax and handling
  - Print
  - Division with Integers
  - Unicode Support, etc.
- Continued Development
  - Python 2(2020)
  - Python 3 (keep going)
- We use Python 3

Python web programming frameworks

- Flask
- Django
- Webapp2, etc.

Outline of today

- Python basics
- Data Structures
- Function

Python Basics

- Variables and types
- Control Flow Tools

Identifier

- *A name that can be used for identifying variable, function, class, object, name etc. in a program, is called an Identifier*

- Must start with upper case (A … Z) or lowercase (a … z) letters
- It can also be started with an underscore ‘_’ , followed by more letters
- It can also be the combination of underscore and numbers

Reserved words

And	Assert	Break
Class	Continue	def
del	elif	else
except	exec	finally
for	from	global
if	import	in
is	lambda	Not
or	pass	print
raise	return	try
while	with	yield

Variables and types

- Python is completely object oriented, and not "statically typed". You do not need to declare variables before using them, or declare their type. Every variable in Python is an object.
- The equal sign (=) is used to assign a value to a variable

```
e.g., a = 1
```

- If a variable is not “defined” (assigned a value), trying to use it will give you an error

- Types: Number; String; List; Set; Dict;etc

Numbers

- Python has simple ways of doing numbers and math
- Numbers: int; float

```
first_int = 8
first_float = 8.0
```

- Declare two variables simultaneously, e.g.

```
a, b = 3, 4
```

Python Interpreter

- Unix shell's search path

```
python3.6
```

- Windows(If installed in C:\python36)

```
set path=%path%;C:\python36
```

- Interactive Mode
- **$** python3.6 Python 3.6 (default, Sep 16 2015, 09:25:04) [GCC 4.8.2] on linux Type "help", "copyright", "credits" or "license" for more information.
- >>>

Python Interpreter

- https://repl.it/languages/python3

Math operations

- The operators +, -, * and / ,**(power), etc

```
>>> first_int + 2
10 (int)
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0 (float)
>>> 8 / 5 # division always returns a floating point number 1.6
>>> first_float – 8 / 6 # 6.4
```

Math operations

```
>>> 5  **  2  # 5 squared
25
>>> 2  **  7  # 2 to the power of 7
128
```

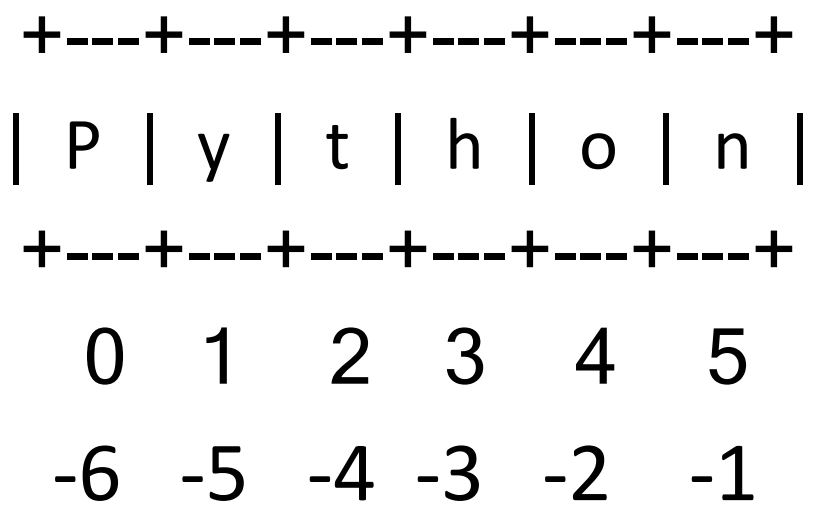
String

- A string is how you make something that your program might give to a human.
- Print them, save to files, send to web servers, etc.
- Double or single quotates

```
e.g., my_string = “hello python !”
                  or ‘hello python !’
```

String index

word = “Python”



Strings access

```
word = 'Python’
word[0] = 'P'
word[5] = 'n'
word[-1] = 'n’
word[-2] = 'o’
word[0:2] = 'Py’
len(s) = 6
```

Mixed variable operation

```
one = 1
two = 2
three = one + two
print(three)  # 3
hello = "hello”
world = "world”
helloworld = hello + " " + world
print(helloworld)  # “hello world”
```

Variables and types

- # This will not work!

```
one = 1
two = 2
hello = "hello”
print(one + two + hello)
```

First script

- Exercise 0: Hello python!

Control Flow Tools

- IF
- FOR
- Range()
- Break and Continue

if Statements

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
        x = 0
        print('Negative changed to zero')
    elif x == 0:
        print('Zero')
    elif x == 1:
        print('Single')
    else:
        print('More')
More
```

“elif” and “else” are optional but recommended

for Statements

```
>>> # Measure some strings:
words = ['cat', 'window', 'defenestrate']
>>> for w in words:
        print(w, len(w))
“cat” 3
“window” 6
“defenestrate” 12
```

for Statements

```
words = ['cat', 'window', 'defenestrate']
>>> for w in words[:]:
        if len(w) > 6:
            words.insert(0, w)
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

The range() Function

```
>>> for i in range(5):
        print(i)
0,1,2,3,4

range(5, 10) # 5 through 9
range(0, 10, 3) # 0, 3, 6, 9
range(-10, -100, -30) # -10, -40, -70
```

The range() Function

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
        print(i, a[i])
0 ‘Mary’ 1 ‘had’ 2 ‘a’ 3 ‘little’ 4 ‘lamb’

>>> list(range(5))
[0, 1, 2, 3, 4]
```

break and *continue* Statements, and *else* Clauses on Loop

- The *break* statement, like in C, breaks out of the smallest enclosing for or while loop.
- The *continue* statement, also borrowed from C, continues with the next iteration of the loop.

break and *continue* Statements, and *else* Clauses on Loop

```
>>> for n in range(2, 10):
        for x in range(2, n):
            if n % x == 0:
                print(n, 'equals', x, '*', n//x)
                break
        else:  # loop fell through without finding a factor

            print(n, 'is a prime number')
```

2 is a prime number
4 equals 2 * 2
6 equals 2 * 3
8 equals 2 * 4
3 is a prime number
5 is a prime number
7 is a prime number
9 equals 3 * 3

break and *continue* Statements, and *else* Clauses on Loop

```
>>> for num in range(2, 10):
        if num % 2 == 0:
            print("Found an even number", num)
            continue
        print("Found a number", num)
```

Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9

pass Statements

```
>>> while True:
      Pass

>>> def initlog(*args):
      pass

>>> class MyEmptyClass:
      pass
```

Exercise 1

- 1a: Factorial
- 1b: Fibonacci $\quad F(n) = \begin{cases} 0, if\ n = 0 \\ 1, if\ n = 0 \\ F(n-1) + F(n-2), if\ n > 1 \end{cases}$

Data Structure

- Lists
- Tuples
- Sets
- Dictionary
- Looping techniques above

List

- List 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.

```
e.g. my_list = [1,2,3]
```

- Lists are very similar to arrays.

List index and access(similar to string)

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

Accessing an index

```
squares[0] = 1
squares[-1] = 25
squares[2:4] = [9, 16]
squares[-3:] = [9, 16, 25]
squares[:] = [1,4,9,16,25]
```

List operation

```
squares = [1, 4, 9, 16, 25]
squares + [36, 49, 64, 81, 100]
# [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
squares = [1, 4, 9, 16, 25]
squares[3] = 64 # [1, 4, 9, 64, 25]
squares.append(216) # [1, 4, 9, 64, 25, 216]
squares[2:5] = [] # [1, 4, 216]
len(squares) # 3
```

List

- Accessing an index which does not exist generates an exception (an error)

List

- It is possible to nest lists (create lists containing other lists), for example:

```
a = ['a', 'b', 'c']
n = [1, 2, 3]
x = [a, n]
x = [['a', 'b', 'c'], [1, 2, 3]]
x[0] = ['a', 'b', 'c']
x[0][1] = 'b'
```

Other List Manipulation

- List.append(x) # append new element to list
- List.extend(L) # add list L to list
- List.insert(I,x) # insert an element
- List.remove(x) # remove an element
- List.pop([i]) # list as stack
- List.clear() # empty list
- List.count(x) # count an element in the list
- List.reverse() # reverse a list
- etc

Tuples

- Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are immutable , and usually contain a heterogeneous sequence of elements that are accessed via unpacking or indexing (or even by attribute in the case of namedtuples). Lists are mutable, and their elements are usually homogeneous and are accessed by iterating over the list.
- E.g.

```
my_tuple=(12345, 54321, 'hello!')
```

Sets

- Python also includes a data type for *sets*. A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Sets

```
>>>basket = {'apple', 'orange', 'apple',
'pear','orange','banana'}
>>>print(basket) # show that duplicates have been removed
    {'orange', 'banana', 'pear', 'apple'}
>>>'orange' in basket # fast membership testing
    True
>>>'crabgrass' in basket
    False
```

Dictionary

- It is best to think of a dictionary as an unordered set of *key:value* pairs, with the requirement that the keys are unique. A pair of braces creates an empty dictionary: {}. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

```
e.g. my_dict = {“first language”:”JS”,
                “second language”:”Python”}
```

Dictionary

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel {'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack'] 4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel {'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> list(tel.keys()) ['irv', 'guido', 'jack']
>>> sorted(tel.keys()) ['guido', 'irv', 'jack']
>>> 'guido' in tel True
>>> 'jack' not in tel False
```

Dictionary generation

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack',
4098)])
    {'sape': 4139, 'jack': 4098, 'guido': 4127}

>>> {x: x**2 for x in (2, 4, 6)}
    {2: 4, 4: 16, 6: 36}

>>> dict(sape=4139, guido=4127, jack=4098)
    {'sape': 4139, 'jack': 4098, 'guido': 4127}
```

Loop

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the
brave'}
>>> for k, v in knights.items():
     print(k, v)

>>> for i, v in enumerate(['tic', 'tac', 'toe']):
       print(i, v)
```

Loop

- For list, tuple or set, their loop methods are similar. E.g

```
my_list = [1,2,3,4]
for item in my_list:
    print(item)
```

The same way for tuple and set.

Exercise 2

- 2a: List filtering
- 2b: Exam scores
- 2c: List and Dictionary operation
- 2d: Data structures

Next

- Functions
- Modules

Function

- ***“A function is a block of reusable, managed code block that is used to perform any certain single operation.”***

- Functions provide better functioning for your application and a high degree of code reusability.

Python Function

- A function block in python begins with a keyword ***def***
- A function name after that keyword ***‘def’*** and parenthese ***‘()’***
- Any input parameter or arguments for a particular functionality should be placed in between these paretheses.
- An optional statement- ***‘fun_docstring’*** is followed by the parentheses, but it’s optional.
- For accessing and invoking a function In python, we should use a colon ***‘:’***
- Then your function or some set of code for performing certain operation.
- At last ***a return***

Function Structure

- # Function: Add

```
def add(a, b):
    print ("Adding : %d + %d" % (a, b))
    return a + b
```

- # Call a function

```
c = add(5, 5)
```

Comments

- Oneline: #

```
# comments
```

- Mulit-line:

```
"""
        comments herer
"""
```

Function arguments

- Default Arguments
- Required Arguments
- Keyword Arguments
- Arbitrary Argument Lists

Default Arguments

- As the name suggests, a default argument always assumes a default value. In case there is no values provided in the function call for a particular argument.

```
def empinfo(name = "ABC", designation):
    print ("Name : ", name)
    print ("Designation : ", designation)
    return
```

empinfo(designation = "Dev")

name: ABC

designation: Dev

Default Arguments

- **Important warning:** The default value is evaluated only once.

```
def f(a, L=[]):
    L.append(a)
    return L
```

print(f(1)) print(f(2)) print(f(3))

[1] [1, 2] [1, 2, 3]

Default Arguments

If you don’t want the default to be shared between subsequent calls, you can write the function like this instead:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

Required Arguments

- Required arguments are those- which are passed to a function in a proper positional / hierarchical order. In python, the number of arguments in the function call should be exactly same as the function definition.

```
def callme( str ):
#This prints a passed string into this function
    print (str)
    return
callme()
```

TypeError: callme() missing 1 required positional argument: 'str'

Keyword Arguments

- Keyword arguments are somehow similar to the function calls. Whenever you use keyword arguments in a function call, it is identified by parameter name directly.
- It allows you to skip the arguments.
- Even you can place the arguments out of order.
- You can also make keyword calls *(using function name)*.

Keyword Arguments

```
def callme( str ):
    print (str)
    return
callme( str = "Calling that function..");
```

Calling that function..

Arbitrary Argument Lists

- All the three above-mentioned arguments are enough to perform any operation. But, still you may need to process a function for more arguments than you specified. In that case you can use this concept. Here comes the concept of Variable Length Arguments.

```
def callme( arg, *vartuple ):
    print ("India")
    print (arg)
    for var in vartuple:
       print (var)
    return
```

callme("Democracy", "or", "Gerontocracy")

India Democracy or Gerontocracy

Python Anonymous Function

- An anonymous function is a function which doesn't require any name or we can say that it can be defined without using a def keyword, unlike normal function creation in the python.
- In python anonymous function is defined using 'Lambda' keyword, that's why it is also called a Lambda function.

```
find_val = lambda x : x*x+2
print (find_val(2))
```

6

- Now, there might be a question blinking in your mind, what is the real use of lambda / anonymous function in Python, as we have already general functions available (using def keyword). Actually in python we do use anonymous function when we require a nameless for a short span of time.

Python Anonymous Function

- An anonymous function can not be directly call to the print, because unlike normal function lambda requires expression.
- It can take any number of the arguments but returns only one value in form of expression.

```
f = lambda x, y, z: x + y +z
```

- Anonymous functions have their own local namespace.
- Anonymous functions can’t access variables other than those in their functioning parenthesis.

Python Anonymous Function

```
lower = (lambda x, y: x if x < y else y)
f = lambda x, y, z: x + y +z
x = (lambda a="fee", b="fie", c="foe": a + b + c)
```

Python Anonymous Function

```
L = [lambda x: x**2,
     lambda x: x**3,
     lambda x: x**4]
for f in L:
    print(f(2))      # prints 4, 8, 16
print(L[0](3))       # prints 9
```

Python Built-In Functions

- The **Filter()** function

  filter(function, sequence)

  offers an elegant way to filter out all the elements of a sequence "sequence", for which the function *function* returns True.

```
pos_int = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_num = list(filter(lambda x: (x%2 == 0,pos_int))
print(even_num)
```

[2, 4, 6, 8, 10]

Python Built-In Functions

- **Map()** Function

```
pos_int = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_num = list(map(lambda x: (x%2 == 0) , pos_int))
print(even_num)
```

[False, True, False, True, False, True, False, True, False, True]

Exercise 3

- 3a: Fibonacci function
- 3b: Sorting
- 3c: Reverse string