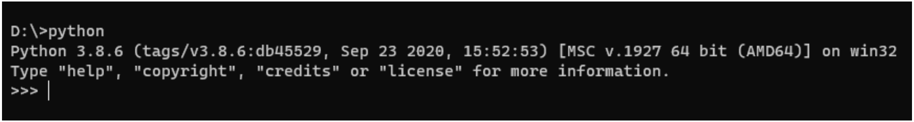
Python Tutorial

Vineel Kovvuri

Table of contents

Introduction

- Python is a dynamically typed programming language invented in 1991 by Guido van Rossum
- Current version is 3.x
- Download python from https://www.python.org/downloads/
- Documentation https://docs.python.org/3/



```
D:\>python
Python 3.8.6 (tags/v3.8.6:db45529, Sep 23 2020, 15:52:53) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> |
```

Figure: Python prompt(REPL) after successful installation

Basics

- Python is not a curly braced language. Each code block is based on indentation
- Python script/program do not contain main function
- Python statements do not end with semicolon

Data Types

Data Type	Comment
Integer	Represents numbers positive and negative. Ex: 100, -20 etc
Float	Represents floating point numbers. Ex: 10.10, -2.120 etc
Boolean	Represents True or False
Strings	Represents strings. Ex: "Hello" or 'World'
None	Represents empty
Lists	A dynamically grow-able collection of objects. Represented with []. Ex: [10, 20, 11]
Tuples	A non grow-able(immutable) collection of objects. Represented with (). Ex: (10, 20, 11)
Dictionary	A dynamically grow-able collection of key-value objects. Represented with {}. Ex: {'name': 'Bob', 'age': 40}

Operators

Type	Operators
Arithmetic Operators	+ − * / // % **
Assignment Operators	= += −= /= //= %= **=
Relational Operators	== < <= > >=
Logical Operators	**and or not**
Membership Operators	**in not in**

Table: Operators

- Unlike C, / operator performs floating point division. $9/2 = 4.5$
- // performs integer division. $9//2 = 4$
- Python support power operator(**). $10**2 = 10^2 = 100$
- Python support multiple assignment. Ex: x, y = 10, 20 or x, y = y, x swaps both x and y
- Python do not support increment(i++)/decrement(i–) operators

Strings

Syntax

```
name = "Riya"
print(name) # prints "Riya Vihaan"
```

Operation	Code	Output
Concatenation	`name = name + " Vihaan"`	Riya Vihaan
Indexing	`name[2]`	y
Slicing	`name[1:3]`	iy
Lower Case	`name.lower()`	riya
Upper Case	`name.upper()`	RIYA
Formatting	`f"My name is {name}"`	My name is Riya

Table: String Operations

String methods

S.capitalize()
S.ljust(width [, fill])
S.casefold()
S.lower()
S.center(width [, fill])
S.lstrip([chars])
S.count(sub [, start [, end]])
S.maketrans(x[, y[, z]])
S.encode([encoding [,errors]])
S.partition(sep)
S.endswith(suffix [, start [, end]])
S.replace(old, new [, count])
S.expandtabs([tabsize])
S.rfind(sub [,start [,end]])
S.find(sub [, start [, end]])
S.rindex(sub [, start [, end]])
S.format(fmtstr, *args, **kwargs)
S.rjust(width [, fill])
S.index(sub [, start [, end]])
S.rpartition(sep)
S.isalnum()
S.rsplit([sep[, maxsplit]])
S.isalpha()
S.rstrip([chars])
S.isdecimal()
S.split([sep [,maxsplit]])
S.isdigit()
S.splitlines([keepends])
S.isidentifier()
S.startswith(prefix [, start [, end]])
S.islower()
S.strip([chars])
S.isnumeric()
S.swapcase()
S.isprintable()
S.title()
S.isspace()
S.translate(map)
S.istitle()
S.upper()
S.isupper()
S.zfill(width)
S.join(iterable)

- Each method description can be found with *help(str.methodname)* at python prompt

Lists

Syntax

```
list1 = [10, 20, 8]
print(list1) # prints "[10, 20, 8]"
```

Operation	Code
Concatenation	`list1 = list1 + [40] #[10, 20, 8, 40]`
Indexing	`list1[2]  # 8`
Reverse Indexing	`list1[-1] # 8`

Slicing	`list1[:] ## entire list [10, 20, 8]`
	`list1[1:3] ## [20, 8]`
	`list1[::2] ## [10, 8] increment in 2 steps`

Reverse	`list1[::-1] ## [8, 20, 10]`
Repeat	`list1*2  # [10, 20, 8, 10, 20, 8]`
Sort	`list1.sort() # [8, 10, 20]`
Length	`len(list1) # 3`
Membership	`10 in list1  # True`

Table: List Operations

- All list methods can be explored with *help(list)* at python prompt

Lists Comprehensions

Syntax - Range function

```
# generate a range from 1 to 9
list1 = list(range(1, 10))
# prints "[1, 2, 3, 4, 5, 6, 7, 8, 9]"
print(list1)
```

range() method do not generate list object we need to pass it to list()

Syntax - Comprehensions

```
# generate a list from 1 to 9
list2 = [2 * x for x in list1]
    # prints "[2, 4, 6, 8, 10, 12, 14, 16, 18]"
print(list2)
```

Tuples

Syntax

```
tup1 = (10, 20, 8)
print(tup1) # prints "(10, 20, 8)"
```

Operation	Code	Output
Indexing	`tup1[2]`	8
Reverse Indexing	`tup1[-1]`	8
Slicing	`tup1[1:3]`	(20, 8)
Repeat	`tup1*2`	(10, 20, 8, 10, 20, 8)
Length	`len(tup1)`	3
Membership	`10 in tup1`	True

Table: Tuples Operations

- All tuple methods can be explored with *help(tuple)* at python prompt

Tuple members cannot be modified. Ex: tup1[1] = 30 do not work!

Sets

Syntax

```
set1 = set([10, 20, 8])
set2 = set([10, 30, 40])
```

Operation	Code	Output
Union	set1 | set2	40, 20, 8, 10, 30
Intersection	set1 & set2	10
Difference	set1 ^ set2	40, 8, 20, 30
Add	set1.add(30)	10, 20, 8, 30
Update	set1.add([70, 80])	10, 20, 8, 70, 80
Remove	set1.remove(10)	20, 8

Table: Set Operations

- All set methods can be explored with *help(set)* at python prompt

Set elements cannot be accessed via indexing. Ex: set1[1] do not work!

Dictionaries

Syntax

```
dict1 = {'name':'Riya', 'age':5}
print(dict1) # prints "{'name': 'Riya', 'age': 5}"
```

Operation	Code	Output
Indexing	`dict1["name"]`	Riya
Insert	`dict1["sibling"] = "Vihaan"`	{’name’:’Riya’, ’age’:5, ’sibling’:’Vihaan’}
Length	`len(dict1)`	2
Membership	`'name' in dict1`	True

Table: Dictionary Operations

- All dictionary methods can be explored with *help(dict)* at python prompt

Selection Statements - if

Syntax

```
if condition:
    stmt
```

Example

```
i = 0
if i < 10:
    print("i is less than 10") # prints i is less than 10
```

Selection Statements - if/else

Syntax

```
if condition:
    stmt
else:
    stmt
```

Example

```
i = 0
if i < 10:
    print("i is less than 10") # prints i is less than 10
else
    print("i greater than or equal to 10") # prints i greater
```

Selection Statements - if/elif/else

Syntax

```
if condition:
    stmt
elif condition:
    stmt
else:
    stmt
```

Example

```
i = 0
if i < 10:
    print("i is less than 10") # prints i is less than 10
elif i > 10:
    print("i is greater than 10") # prints i is greater than
else
    print("i is equal to 10") # prints i is equal to 10
```

Selection Statements - if/else ternary

Syntax

```
Z = X if condition else Y
```

Example

```
i = 0 if i < 10 else 10
```

Python's ternary operator == traditional C ternary operator

```
Z = condition ? X : Y;
```

Statements - for loop

Syntax

```
for element in collection:
    stmt
```

Example

```
arr = [10, 5, 41, 6, 8, 3]
for ele in arr:
    print(ele) # prints 10 5 41 6 8 3
```

- It loop over a collection of items like a list/dictionary
- Loop variable(ele) holds one element of the collection in each iteration

Python's for loop != traditional C for loop

```
for (int i = 0; i < n; i++) {}
```

Statements - while loop

Syntax

```
while condition:
    stmt
```

Example

```
i = 0
while i < 10:
    print(i) # prints 0 1 2 3 4 5 6 7 8 9
    i = i + 1
```

- execute the body of the loop until the condition is false

Python's while loop == traditional C for loop

```
for (int i = 0; i < n; i++) {}
```

Statements - Unconditional Jumps

break syntax

```
while i < 10:
    if i % 2 == 0:
        break   # break out of the loop
```

continue syntax

```
while i < 10:
    i = i + 1
    if i % 2 == 0:
        continue
        # continue to next iteration. Skip below statements
    print(i)
```

Statements - pass

Syntax

```
while i < 10:
    i = i + 1
    if i % 2 == 0:
        pass  # Empty statement. It does not do anything!
    print(i)  # this code gets executed
```

Python's pass statement == C's empty statement

```
while (i < 10) {
    if (i % 2 == 0)
        ;                   // python's pass is same as this
}
```

Functions

Syntax

```
def name(arg1, arg2,... argN):
    stmt
```

Example

```
def add(a, b):
    print(a+b)

add(10, 20) # this will print 30
```

Functions - Return Statement

Syntax

```
def name(arg1, arg2,... argN):
    stmt
    return expr
```

Example

```
def add(a, b):
    return a+b

result = add(10, 20)
print(result)
```

Functions - Builtin

Builtin Function	Description
len(seq)	return the size of the collection `len([10, 3, 4, 20]) ## 4`
min(seq)	return the maximum element in a collection `max([10, 3, 4, 20]) ## 20`
max(seq)	return the minimum element in a collection `min([10, 3, 4, 20]) ## 3`
sum(seq)	return the sum of all the elements in a collection `sum([10, 3, 4, 20]) ## 37`

range(start, stop[, step])	generate a collection b/w start and stop
	`range(1, 5) ## generate [1,2,3,4]`
	`range(1, 10, 2) ## generate [1, 3, 5, 7, 9]`

reversed(seq)	reverses a collection `reversed([1,2,3,4]) ## [4,3,2,1]`
sorted(seq, keyfn)	sorts a collection `sorted([3,2,1,4]) ## [1,2,3,4]`
zip(seq, seq...)	zips two separate collections in to one `zip([3,2], [1,2]) ## [(3, 1), (2, 2)]`
filter(fn, seq)	filters elements in a collection based on the predicate `filter(lambda x : x < 10, [11, 21, 1, 8]) ## [1, 8]`
map(fn, seq)	creates a new collection whose elements are mapped according to the function `map(lambda x : x + 10, [1, 2, 4, 8]) ## [11, 12, 14, 18]`

Table: Builtin Functions

Functions - Lambda - Anonymous Functions

Syntax

```
# defines a lambda function
fn = lambda arg1, arg2,... : <expression>
```

Example

```
# defines a lambda function and stores
# in a variable called fn
fn = lambda x : x + 10
# call the lambda function
print(fn(1)) # print 11
```

Example

```
# lambda with two parameters
add = lambda x, y : x + y
print(add(1, 2)) # print 11
```

Functions - Named Arugments

Syntax

```
def name(arg1, arg2):
    stmt
    return expr

name(arg2=10, arg1=20)
```

Example

```
def add(a, b):
    print(f"a is {a} b is {b}")
    return a + b

result = add(b=10, a=20)
# prints "a is 20 b is 10"
print(result)
```

Functions - Default Arguments

Example

```python
# if the argument b is not passed a
# default value of 30 is used
def add(a, b=30):
    print(f"a is {a} b is {b}")
    return a + b

result = add(10)
# prints "a is 10 b is 30"
print(result)
```

Functions - Variable number of arguments

Example

```python
# all arguments passed to this function are
# converted to a tuple
def add(*args):
    res = 0
    for x in args:
        res += x
    return res

# can call the function with different
# number of arguments
print(add(10, 20, 40)) # prints 70
print(add(10, 20, 40, 30)) # prints 100
```

Classes - Basics

Example

```python
class Book:
    # self is similar to this keyword in Java/C#
    # it is not passed during the call.
    def add_book(self, name):
        print(f"book {name} is added")

# create an object of type Book
b = Book()
# invoke a method
b.add_book("Python Programming")
```

Classes - Constructor

Example

```python
class Book:
    # __init__ is a builtin method which
    # represents the constructor
    def __init__(self, name):
        self.name = name

    def get_book_name(self):
        return self.name

# create an object of type Book
b = Book("Python Programming")
# invoke a method
print(b.get_book_name())
```

Classes - __str__

Example

```python
class Book:
    def __init__(self, name):
        self.name = name

    # __str__ is a builtin method which is
    # similar to tostring() method in Java/C#
    def __str__(self):
        return f"Book name is {self.name}"

# create an object of type Book
b = Book("Python Programming")
print(b)
```

Libraries - Reading text files

Example

```
file = open(r"C:\abc.txt", "r")
str = file.readline()

while str:
    print(str)
    str = file.readline()

file.close()
```