Python syntax

Andy Kim

Basic Python Syntax

- VScode shortcuts

  - ctrl+space (vscode): command template
  - ctrl+shift+p (vscode): command palette
  - shift+enter (vscode): tests the current line in terminal / works w multiple line selection
  - "". # will show many functions associated with string class (vscode)
  - softtab: converts tab to 4 spaces (vscode)

- variables are objects   a = 100; [a] refers to 100 in memory; id(a): address

- mystring = 'hello world'; mystring refers to an array (string instance); "text" = 'text'

  ```
  mystring[1:4]: 'ello'
  mystring.split('\<divider\>'): generates a list
  ```

- indent in Python means a code block

- a = 'At'; b='work'; a+'<space>'+b returns 'At work'

- List:

  ```
  a = [1, 2, 3, 4]
  a[0]: 1
  a[-1]: 4
  a[0:2]: [1, 2, 3]
  a.append(5): [1, 2, 3, 4, 5]
  a.insert(3, 3.5): [1, 2, 3, 3.5, 4, 5]
  len(a): 6
  del a[-1]: [1, 2, 3, 3.5, 4]
  ```

- Tuple: similar to list, but cannot be modified and is faster

```
t = (1, 2, 3)
t[1]: 2
# all functions are the same in slicing and indexing as list
```

- Dictionary

```
price = { } # generates an empty dictionary
price['a'] = 100 # price: {'a':100}
price['b'] = 10 # price: {'a': 100, 'b':10}
price['a']: 100
price.keys(); price.values() returns keys and values
keys = list(price.keys()) # converts to list
values = list(price.values()
'a' in price.keys() # True
```

- Bool: == != > < and or not True False

- if <exp>: elif <exp>: else:

- c style print

```
print("%s ..." %var) # no comma before %var
```

- in operator

```
for i in [0, 1, 2, 3, 4]:
    print(i)
price_dict = {'a': 100, 'b':10, 'c':1}
for a, b in price_dict.items():
    print("%s: buy %s" %(a, b))
for a in price_dict.keys():
    print("%s: buy %s" %(a, price_dict[a]))
```

- list(listrange(1, 10)) returns [1, 2, ..., 9]

- 2nd order list

```
A = [[101, 102, 103], [201, 202, 203]]
A[0]: [101, 102, 103]
A[0][1]: 102
for i in A:
    for j in i:
        print(j) # 101, 102, 103 ...
```

- print("") includes \n; to prevent use print("string", end="")

- del var deletes defined variable

- Function

  ```
  def func(argv): # argv has internal scope
      pass # null operation
      return xx # any type
  ```

- import

  ```
  import \<directory.modulefilename\> # filename does not need .py
  # C-code module could be also imported (refer to external sources)


  from os import listdir # only imports listdir
  from os import *
  import os as linuxos # alias
  ```

- import time; time.time(); time.ctime(); time.sleep(1)

- dir(time): shows functions in the module time (in console mode)

- import os

  ```
  os.getcwd(): current working space
  os.listdir('path')
  files = os.listdir('path')
  len(files) # returns the number of files
  for x in files:
      if x.endswith('exe')
          print(x)
  ```

- "_" latest return value

- naming rule

  - camel case: Class (camel case starting with small letter is not used in Python)
  - snake case: with () - function, without () - variables

- string cut

  - "string"[a:b]: a starts with 0, b is not included
  - either a or b can be omitted; a and/or b can be negative numbers

- len(): prints length of strings

- type(): prints type str, int, float (no long in C)

- 0 is not equal to 0.0

- %: residual operator

- //: integer divide (e.g., $5/2 = 2.5$, $5//2 = 2$)

- **: power (e.g., $5**4 = 5^4$)

- Variable declare in python: no need to specify type

- +=, -=, *=, /=, %=, **=

- +=, *=: works with strings as well

- input("message:"): takes user input from screen

- converting string to number: int(), float()

- str(): converts int to string

- format function: "{} {}".format(10, 20) # "10 20"

- string class functions:

    - strip() / lstrip() / rstrip(): removes spaces in both/left/right
    - rstrip('\n'): removes '\n' from the string
    - is...()
    - find()/rfind() # returns index starting from index 0
    - "he" in "hello" # True or False
    - "10 20 30".split(" ") # ['10', '20', '30']

- boolean in python: True / False (starts with upper case)

- date and time

    ```
    import datetime
    now = datetime.datetime.now()
    now.year
    now.month ...
    ```

- double condition

4

```
    if 3 < a < 10:
...
```

- multi line input example:

```
    if 1 <= a <=3 or \
        101 <= a <=103 or \
        201 <= a <=203:
        print("a is in a specific condition")
```

- using "in" operator example:

```
    b = input("b> ") # b is a string instance, can be handled like a list
    if b[-1] in "02468":
        print("b is a even number")
```

- if number % 2 == 0: # faster and more efficient method than the above

- list: [index], + * operator possible, like handling string

- list can include various kinds of types [1, 'str', 2, True]

- list operators:

    – append(last element); insert(target index, value); extend(another list)
    – pop(index #last if none); clear() # remove all; del list_name[index]
    – "in" operator works with list
    – remove(value) # remove first encounter of the value from the list

- debug: breakpoint - F9

- dictionary type

    – {:, ... } # curly bracket in definition
    – dict_a["key name"] # bracket for index
    – dict_a["new name"] = value # append
    – del dict_a["key name"]
    – key in dict_a # check if exists
    – dict_a.get("key") # returns None, if key does not exist

- range(i, j, inc) # returns range class instance; inc: increment; from i to j-1

- range(0): nothing

- sorted(list or other iterable) returns a sorted list or iterable

- while / break / continue

- two types of functions: destructive and nondestructive functions

```
new_string = old_string.split(" ") # nondestructive
list.append('new element') # destructive
```

- string addition

```
("string\n" "string" ...\) # same as "string\nstring"
```

- string join:

```
"\n".join(["string1", "string2", "string3"]) # same as string1\nstring2\nstring3
```

- import textwrap; textwrap.dedent(""""Strings..."""") # removes indents (or spaces) from the beginning of each line

- reversed(list) # returns the reversed sequence of the list / but only once

- enumerate in list and dictionary; enumerate(list, tuple, string) returns index and data; convenient when looking for data in a list or iterable

```
example_list = [1, 2, 3, 4, 5]
for index, element in enumerate(example_list):
    print("{}th element is {}".format(index, element))


example_dict = {
    "key A": "val A",
    "key B": "val B",
    "key C": "val C",
    "key D": "val D",
}
for key, element in example_dict.items():
    print("{}th element is {}".format(key, element))


"{:3}".format(i) # puts i into {} with three digits/spaces
```

- array creation:

  - array = [i*i for i in range (0, 20, 2) if 100 < (i*i) <300]
  - arr = ["a", "b", "c", "d", "e"]

- new_arr = [str for str in arr if str != "c"]
- can use function within [or] to create a list

- function arguments - def funtion(a, b, var1=10, var2=20, *values):

  - a, b # required arguments
  - var1, var2 # default arguments
  - *values # tuple variable for the rest of the arguments

- recursion function: must create a exit mechanism / recursion is not efficient

- to improve speed of recursion, memorize calculation result in a dictionary

- to use global variable in a function, use "global" keyword e.g., global var

- tuple with single element: (1,)

- tuple can be used without ():

  ```
  a, b = 10, 20 # a = 10, b = 20
  a, b = func() # def func(): return 10, 20
  ```

- in Python, functions can get functions as arguments

- map and filter functions

  ```
  list_map = [1, 2, 3, 4, 5]
  def power(x):
      return x**2
  list_after_map = map(power, list_map)
  print(list_after_map) # shows object address
  print(list(list_after_map)) # type casting required

  def under_3(x):  # returns True / False
      return x < 3
  list_after_filter = filter(under_3, list_map)
  print(list(list_after_filter)) # casting required
  ```

- lambda function

  ```
  list_a = [1, 2, 3, 4, 5]
  a = map(lambda x: x*x, list_a)
  b = filter(lambda x: x<3, list_a)
      print(list(a))
      print(list(b))
  ```

- file open and close

```
file = open("XXX.py", mode="r", encoding="UTF-8")
i = 0
for line in file:  # could access each line in file with this expression
    print(i, line, end="")
    i += 1
print(file.read())
file.close()
with open("XXX.py", mode="r", encoding="UTF-8") as file:
    print(file.read())   # no need to call close() when used with-open
```

- file read/write/append mode: has to be specified with one mode

- exception handling

```
try:
    <code> # try-exception can be used more than just catching exceptions
except Exception as e:
    pass <code for when exception occurs>
else:
    <code for when no exception occurs>
finally:
    <code to be executed in any case /
    especially useful if try contains function return>
```

- exception advanced; check always the exception name for the following usage

```
try: ...
except IndexError as exception:
    ...
except ValueError as exception:
    ...
except Exception as exception:
    ... # everything else
```

- if statement: 0, 0.0, None, empty container (string, byte, list, tuple, dict) are considered as False

- pass vs raise NotImplementedError

- raising exception for debugging/developing

```
raise <Exception Instance>  # usually ends with Error;
```

```
    Exception_Instance("message") # casting into instance from class
    try:
        raise NotADirectoryError("Message")
    except NotADirectoryError as error:
        print(error)  # prints message without terminating program
```

- finally keyword:

```
    try:
        break; return; # when the code escapes by break or return
    except:
        pass
    finally:
        statements here must be executed regardless with break/return
        e.g., file close / db close / etc...
        when there are many returns, finally: could make code simpler
```

- import

```
    import math
    import math as m
    from math import pi, sin
    from math import pi as p, sin as sine
```

- do not make file name as random.py: it conflitcs with python keywords

- refer to the python.org documentation

- modules

```
    import sys
    print(sys.argv)
    import os
    os.system("<linux command>")
    import time
    time.sleep(5) # sleeps for 5 sec
    from urllib import request # url = unique resource location used in web
    target = request.urlopen("https://google.com")
    output = target.read()
    print(output) # if output starts with b', it means it is binary code
```

- tensor: group of vectors / 0th tensor: scaler / 1st tensor: vector

- example:

```
import datetime
now = datetime.datetime.now() # identical with below
from datetime import datetime
now = datetime.now()
```

- text file open

```
f = open ("path to file", "rt")  # r: read, w: write, t: text
lines = f.readliens()
for line in lines
    print(line)
    nline = line.split("\n")[0]
    print(nline)
f.write("xxx\n")
f.close()
```

- binary file open

```
file = open("filename.xxx", "rb") # b keyword for binary reading
content = file.read()
file.close()
print(type(content))
file = open("output.png", "wb") # binary file handing has to specify "b"
file.write(content)
file.close()
```

- framework: in framework, ext modules are executing user created python files.

  – Flask, Django: frameworks

- framework <-> library

- importing/creating modules: make separate files and put them in the same folder

- Main module vs imported

```
__name__ == "__main__"
# True if run independently / if imported, __name__ is the imported module name
# checks if the current file is imported or executed
# usually used when testing module's functionality
```

- Class

```
class A:
    def f1():
    def f2(self):


A.f1()  # OK; class A works as a namespace
a = A()
a.f1()  # error
a.f2()  # OK; instance always receives 'self' arg
A.f2(a) # also OK
```

Class is a namespace; variables/methods are stored as dictionary in the namespace

```
A.__dict__ # listing dictionary values
a1 = A()
a2 = A()


A.market = 's'
a1.market = 's1'


a1.market  #s1
a2.market  #s
```

Class inheritance

```
class A:
    def f1(self):
        pass


class B(A): # Inheritance
    pass


b = B()
b.f1() # OK
```

Constructor / Destructor

```
__init__ # constructor
__del__ # destructor


__init__(self, ...)
# self: use as the first argument for class intance functions/methods
```

```
    bool_value = isinstance(instance, Class)

    __str__(self) # lets use str(<instance>) / python developer agreement

    __eq__(self, value), __le__(self, value), ... # operator overloading
```

* Class variable: <Class>.variable

* Class method:

```
    Class A:
        @classmethod
        def class_method(cls):
            print("...")
    A.class_method() # a way to access a class method
    __del__(self) # destructor
```

- garbage collector: automated in python
- list generation '" list_with_10_zeros = [0 for n in range(10)] list_with_power_of_two = [n**2 for n in range(0,10)] '"
- enumerating alphabets '" for i in range(0,26): print(str(chr(i+65))) # ASCII 65 is A print(str(chr(i+ord('A')))) # ord('<char>') returns ASCII value '"

Korean Encoding: utf-8 or euc-kr

Regular Expression

VScode intellisense icons

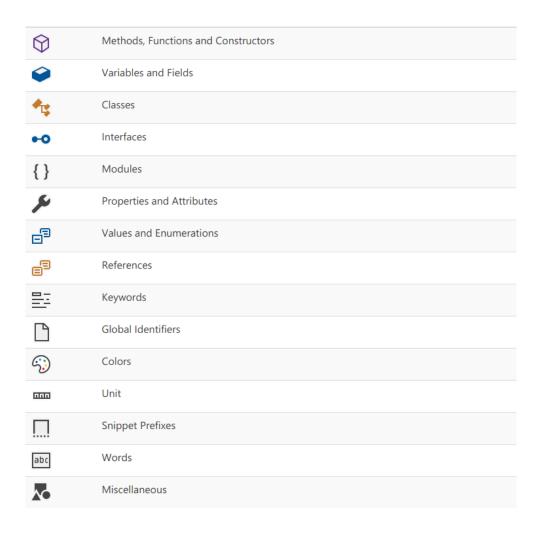|	
⬡	Methods, Functions and Constructors
⬢	Variables and Fields
⬥	Classes
⊶	Interfaces
{ }	Modules
🔧	Properties and Attributes
⬚	Values and Enumerations
⬚	References
☰	Keywords
📄	Global Identifiers
🎨	Colors
▭	Unit
▢	Snippet Prefixes
abc	Words
◣	Miscellaneous

Figure 1: vscode intellisense icons