Data Science for Civil Engineering

Introduction to Python

Jintao Ke

Department of Civil Engineering
Faculty of Engineering
The University of Hong Kong, Pokfulam, Hong Kong
Email: kejintao@hku.hk

Table of Contents

Learning Outcomes

- Understand features, advantages, and programming environment configuration in Python
- Understand the basic syntax of Python, such as data types, basic operations, containers, etc
- Comprehend the Flow Control Statements in Python, and learn how to define the function and class and call them

The origin of Python

- Guido van Rossum started Python in late 1989, and the first public release of Python appeared in early 1991.

Figure: Name of Python: BBC comedy series: “Monty Python‘s Flying Circus”

Figure: Guido van Rossum, A Dutch programmer

Why use Python?

- Python has been increasingly prevalent among all types of programming languages

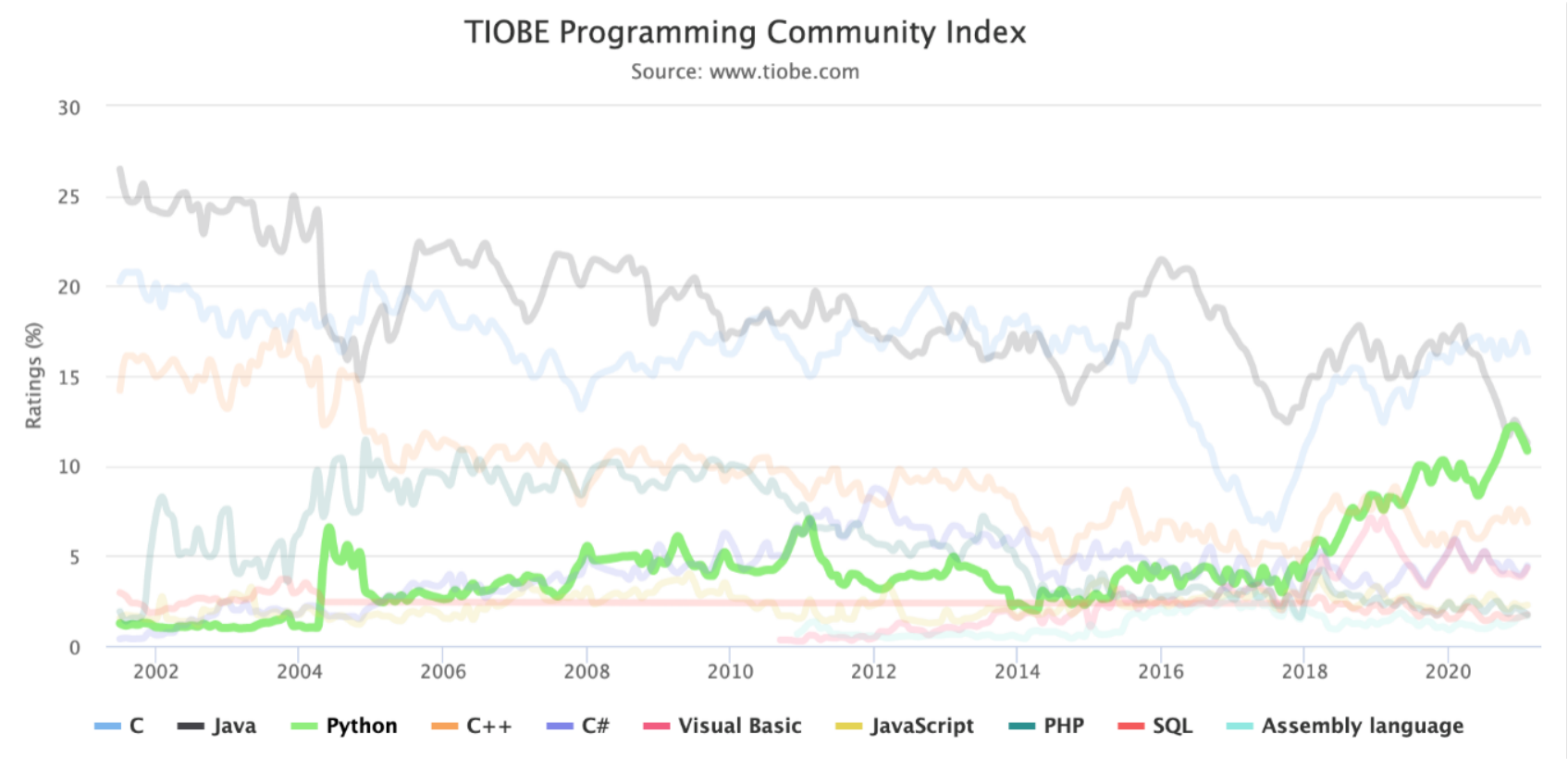
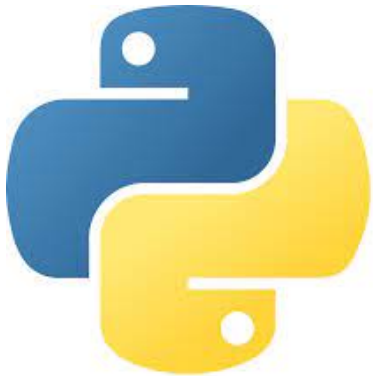


Figure: Popularity of Python Language

Why use Python?

Python is a powerful tool in:

- Data Analytics
- Machine Learning
- Software Development
- Education
- Webpage Developing
- Commercial Applications



Why use Python?

Python is also an appropriate one for beginners because of these reasons:

- User-friendly
- Highly compatible(able to work with other programming languages)
- Well-resourced community
- Plenty of tutorials and complete toolkits from third parties
- No compiling, platform-independent
- Simple code and grammer with high readability

Interpreted Language VS Compiled Language

- Python is an interpreted language and C++ is a typical compiled language

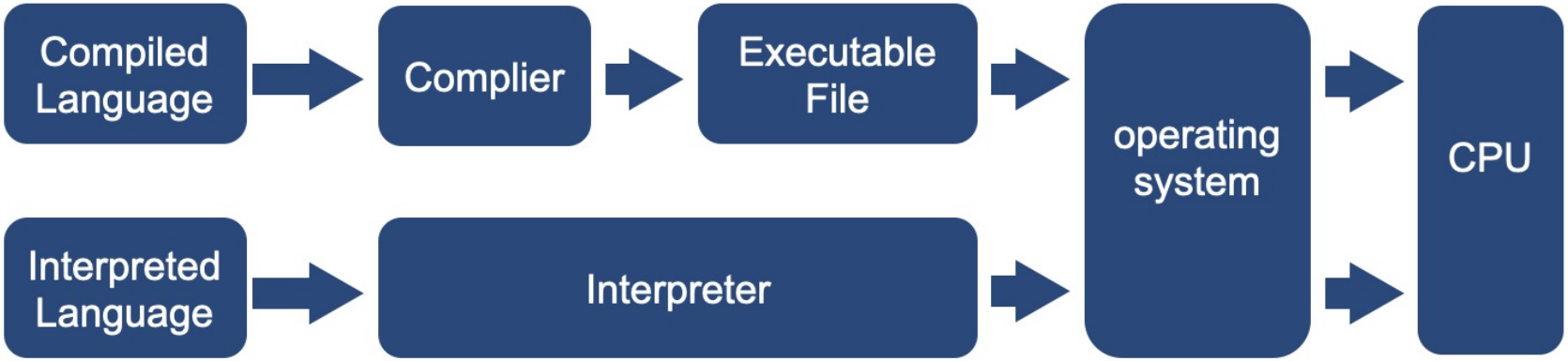


Figure: Working Mechanism of two Languages

The advantages of Python as an interpreted language:

- No compilation required and simple execution

A quick start for Python

- Anaconda is an open source Python distribution for managing toolkits, you can use it to start Python programming immediately[1]

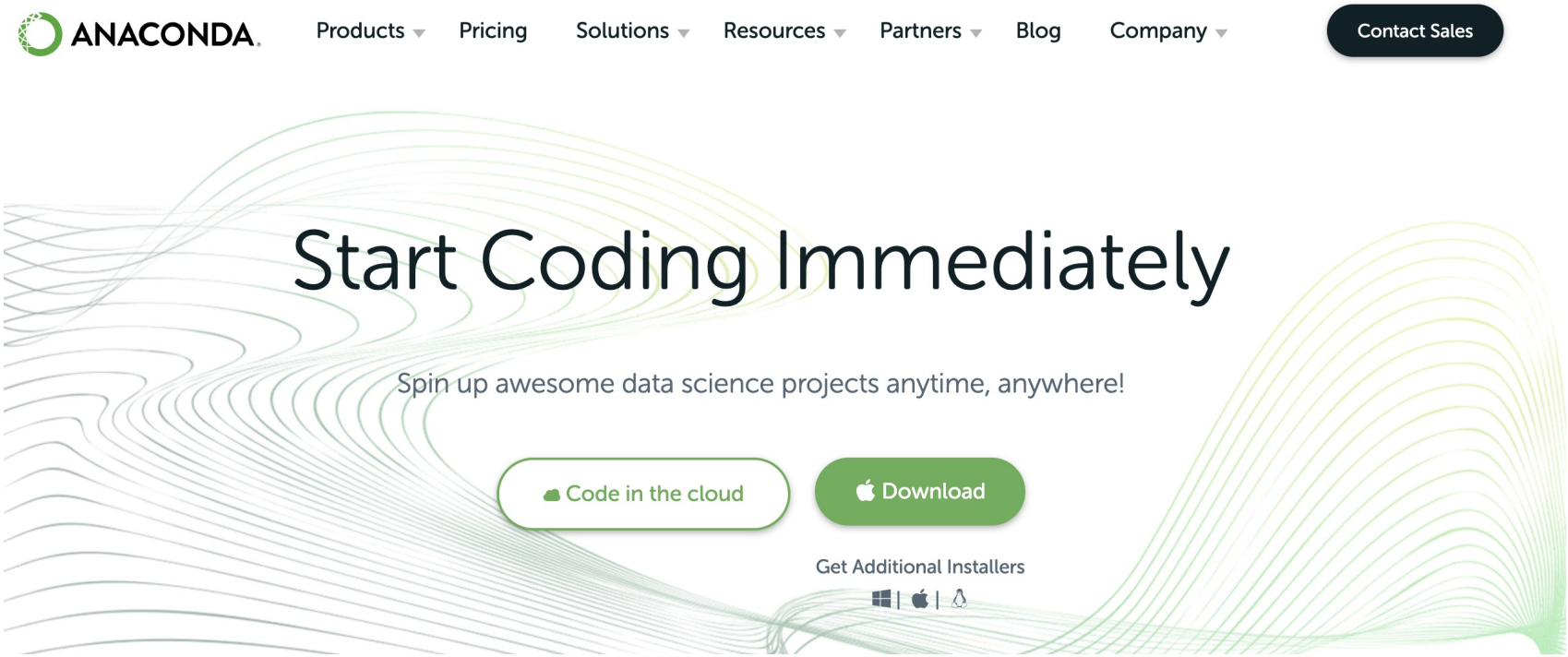
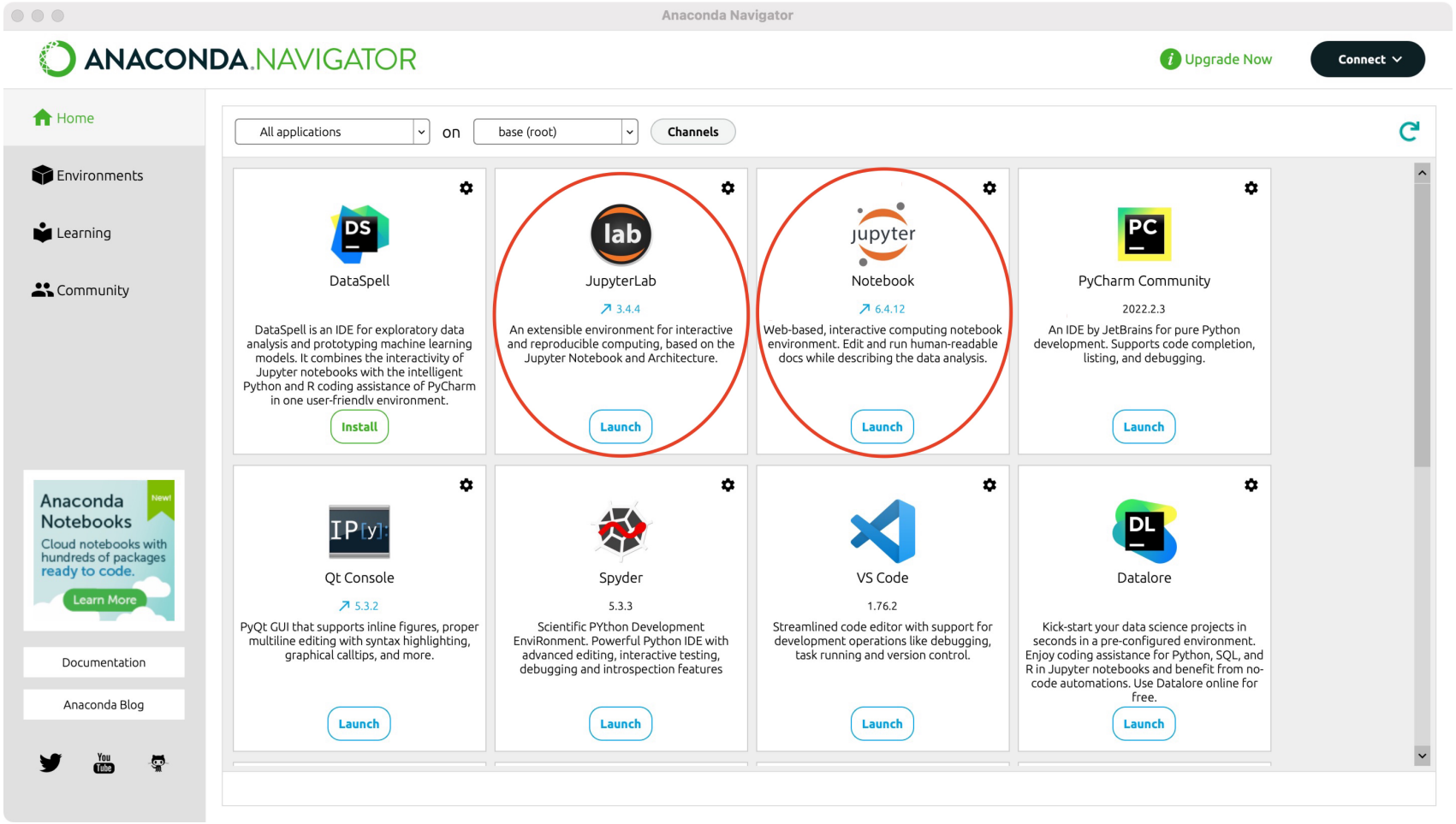


Figure: Anaconda

[1] Relevant details will be introduced by TA in the tutorials.

A quick start for Python

- In Anaconda, you can launch Jupyter Notebook or JupyterLab(more recommended) to start you journey in Python



Outline

Variables in Python

- Variable
  - Specific, case-sensitive name
  - Call up value through variable name

```
dog_name = 'Bella'
dog_age = 1
dog_birth_year = 2022
dog_weight = 28.3
vaccinated = True
```

Figure: Assign values to variables

- In Python, we do not have to explicitly declaim the type of the variable (e.g., 'dog_name') in any way
- Python automatically detected the type of a variable

Variable Types

There are some commonly used types in Python:

- Integer type
- Float type
- String type
- Boolean type
- Null type(None)

```
print('Type of dog_name',type(dog_name))
print('Type of dog_age',type(dog_age))
print('Type of dog_weight',type(dog_weight))
print('Type of vaccinated',type(vaccinated))
```

✓ 0.0s

```
Type of dog_name <class 'str'>
Type of dog_age <class 'int'>
Type of dog_weight <class 'float'>
Type of vaccinated <class 'bool'>
```

- The variable name consists of letters, digits, and underscores, and it cannot start with a digit, e.g., dog_name1
- Constant variable are usually written in uppercase, e.g., PI = 3.14

Variable Types

Reflection Question:

What is the difference between a&b?

```
a = 100
b = str(a)
print('a:',a)
print('b:',b)
```

✓ 0.0s

```
a: 100
b: 100
```

Variable Types

Reflection Question:

- What is the difference between a&b?

```
a = 100
b = str(a)
print('a:',a)
print('b:',b)
```

✓ 0.0s

```
a: 100
b: 100
```

- Actually, a is an integer number(a value), and b is a string consists of 3 chacaters: '1', '0', '0'.
- String variables cannot be evaluated numerically, so you may transform them when needed.

Outline

Arithmetic Operations in Python

Operator	Operation	Example	Result
=	Assign a value to a variable	a = 2	# a = 2
+	Addition	a = 3 + 5	# a = 8
-	Subtraction	a = 5 - 3	# a = 2
*	Multiplication	a = 2 * 3	# a = 6
/	Division(it always returns a float)	a = 15 / 2	# a = 7.5
//	Floor division (or integer division)	a = 15 // 2	# a = 7
**	Power	a = 4 ** 3	# a = 64
%	Remainder	a = 15 % 2	# a = 1

String Operations in Python

Create a string:

- A string is a sequence of characters
- The characters are specified either between single(‘) or double (“) quotes
- E.g.,a = “abc”or a = ‘abc’

```
message = 'Hello World!'
print(message)
message = "Hello World!"
print(message)
```

✓ 0.0s

```
Hello World!
Hello World!
```

String Operations in Python

Concatenate strings:

- you can directly use '+' operator
- Another way is to use 'join'

```
message1 = 'Hello'
message2 = ' World!'
message = message1 + message2
print(message)

message = ('').join([message1,message2])
print(message)
```

✓ 0.0s

```
Hello World!
Hello World!
```

String Operations in Python

- Repeat a string

```
message = 'Hello World!'
print(message*3)
```

✓ 0.0s

```
Hello World!Hello World!Hello World!
```

.split()

- Split a string

```
# split it by space
words = message.split(' ')
print(words)
```

✓ 0.0s

```
['Hello', 'World!']
```

String Operations in Python

.replace()

- Replace a word with another one

```
new_message = message.replace('World','everyone')
print(new_message)
```

✓ 0.0s

```
Hello everyone!
```

.upper()/.lower()

- Upper, turning all the characters into uppercase, you can use 'lower' to do the opposite operation

```
message = 'Hello World!'
message_upper = message.upper()
print(message_upper)
```

✓ 0.0s

```
HELLO WORLD!
```

String Operations in Python

.capitalize()

- Capitalize, turning the first character to uppercase, left all the others be lowercase

```
message = 'Hello World!'
message_capitalize = message.capitalize()
print(message_capitalize)
```

✓ 0.0s

```
Hello world!
```

.title()

- Title, making the first character of each word be uppercase, and let all the others in each word be lowercase

```
message = 'hello world!'
message_title = message.title()
print(message_title)
```

✓ 0.0s

```
Hello World!
```

Outline

Lists

- Probably the most fundamental data structure in Python
- A list can be generated with [], containing arbitrary number of elements that stored in sequential order
- Indexing begins from 0
- The elements don't need to be of the same type.
- The elements of a list can be any type, including "List" itself
- List is mutable(the value can be changed).

```
list = [1,'b',3.5,'dog',True]
```

Figure: An example of List

Operation of Lists

- Starting with 0, an element in list can be called using index

```
list = [1,'b',3.5,'dog',True]
print('list[0]:',list[0])
print('list[3]:',list[3])
```

✓ 0.0s

```
list[0]: 1
list[3]: dog
```

- Similarly, you can start from the end to call an element, counting from -1, -2 to the beginning one

```
list = [1,'b',3.5,'dog',True]
print('list[-1]:',list[-1])
print('list[-3]:',list[-3])
```

✓ 0.0s

```
list[-1]: True
list[-3]: 3.5
```

Operation of Lists

- An element in a list can be changed by assigning a new value

```
list = [1,'b',3.5,'dog',True]
list[0] = 3
print('List:',list)
```

✓ 0.0s

```
List: [3, 'b', 3.5, 'dog', True]
```

- Using '.insert()' function to insert an element into the specified position with index

```
list = [1,'b',3.5,'dog',True]
list.insert(3,'toy')
print('List:',list)
```

✓ 0.0s

```
List: [1, 'b', 3.5, 'toy', 'dog', True]
```

Operation of Lists

- You can use '.append()' function to add a new value at the end of a list

```
list = [1,'b',3.5,'dog',True]
list.append(2)
print('List:',list)
```

✓ 0.0s

```
List: [1, 'b', 3.5, 'dog', True, 2]
```

- You can use '.pop()' function to delete a value at the end of a list

```
list = [1,'b',3.5,'dog',True]
list.pop()
print('List:',list)
```

✓ 0.0s

```
List: [1, 'b', 3.5, 'dog']
```

Operation of Lists

Slicing a list with square brackets

- Get or set elements in a list by slicing with square brackets
- General format for slicing: list[start: end: step]
  - The element of start index is inclusive
  - The element of end index is not inclusive
  - 'Step' represents step length, you can omit the step and it will be 1

```
list = [1,'b',3.5,'dog',True,4,7]
print('list[0:4]:',list[0:4])
print('list[0:4:2]:',list[0:4:2])
print('list[2:5]:',list[2:5])
```

✓ 0.0s

```
list[0:4]: [1, 'b', 3.5, 'dog']
list[0:4:2]: [1, 3.5]
list[2:5]: [3.5, 'dog', True]
```

Sorting Operation of Lists

.sort()

- it modifies the original list

```
list = [5,7,2,4,1,9]
list.sort()
print('List:',list)
```

✓ 0.0s

```
List: [1, 2, 4, 5, 7, 9]
```

sorted()

- it returns a new sorted list and leaves the original list unchanged

```
list = [5,7,2,4,1,9]
print('sorted(List):',sorted(list))
print('List:',list)
```

✓ 0.0s

```
sorted(List): [1, 2, 4, 5, 7, 9]
List: [5, 7, 2, 4, 1, 9]
```

Tuples

- Tuples are similar to lists, but they are immutable
- Pretty much anything you can do to a list that doesn't involve modifying it, you can do to a tuple
- Tuples are defined by using parentheses (or nothing) instead of square brackets.

```
tuples = ('a','b',1,2)
print('type(tuples):',type(tuples))
print('tuples:',tuples)
print('tuples[1]:',tuples[1])
```

✓ 0.0s

```
type(tuples): <class 'tuple'>
tuples: ('a', 'b', 1, 2)
tuples[1]: b
```

Figure: An example of Tuple

Tuples

- Tuples seem inconvenient, and why do we need it?
- In defining a function(will be introduced later), tuple is important to return a reliable result.

```
def sum_and_product(a,b):
    summing = a + b
    product = a*b
    return summing,product

s, p = sum_and_product(1,2)
print('sum:',s)
print('product:',p)
```

✓ 0.0s

```
sum: 3
product: 2
```

Figure: Tuple in function

Dictionaries

- Dictionary associates values with keys
- Dictionary can be created by listing the comma separated key-value pairs in curly brackets. Keys and values are separated by a colon
- Dictionary allows to quickly retrieve the value corresponding to a given key using square brackets

```
students_grades = {"John":85,"Amy":87,"David":90,"Lisa":88}
print("David's Grade:",students_grades["David"])
```

✓ 0.0s

```
David's Grade: 90
```

Figure: An example of Dictionary

Sets

- Set represents a collection of unique elements
- Set can be created by listing its elements between curly brackets.
- Set can also be created by using set() itself

```
a = set('aaaaabbbbbcccc')
b = {1,1,2,2,3,3}
print('a:',a)
print('b:',b)
```

✓ 0.0s

```
a: {'a', 'b', 'c'}
b: {1, 2, 3}
```

Figure: Two methods to generate a set

Role of Sets

Fast menbership checking:

- 'in' is a very fast operation on sets

```
import time
print('Time usage for list and set:')
word_list = [1,2,3,4,5,6]
%timeit 4 in word_list

word_set = {1,2,3,4,5,6}
%timeit 4 in word_set
```

✓ 7.1s

```
Time usage for list and set:
56.2 ns ± 0.475 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
30.9 ns ± 0.189 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

Role of Sets

Unique items searching:

- You can turn a list into a set to check the number of unique values

```
a = [1,3,4,4,2,2,1,5,7,1]
b = set(a)
number = len(b) #len can be used to get the length of a sequence
print('set(a):',b)
print('the number of unique values in a:',number)
```

✓ 0.0s

```
set(a): {1, 2, 3, 4, 5, 7}
the number of unique values in a: 6
```

Operation of Sets

A set is mutable

- You can use '.add()' function to add a value to a set
- You can use '.remove()' function to delete a value in a set
- if you add an existing value in a set, the set will keep unchanged

```
a = {1,1,2,2,3,3}
print('set(a):',a)

a.add(4)
print('set(a):',a)
a.add(4) #existing value
print('set(a):',a)

a.remove(1)
print('set(a):',a)
```

✓ 0.0s

```
set(a): {1, 2, 3}
set(a): {1, 2, 3, 4}
set(a): {1, 2, 3, 4}
set(a): {2, 3, 4}
```

Summary: Data Structures in Python

Type	Example	Description	Changeability	Order
List	[1,2,3]	List: an mutable, ordered collection	mutable	ordered
Tuple	(1,2,3)	Tuple: an immutable, ordered collection	immutable	ordered
Set	{1,2,3}	Set: Unordered collection of unique values	mutable	unordered
Dict	{'a':1, 'b':2, 'c':3}	Dictionary: Unordered (key, value) mapping	mutable	unordered

Conditional Judgment Statement(if-else)

- The if-else statement is used to decide whether a certain statement or block of statements will be executed or not

```
if condition1:
    statement1_1
    statement1_2
    ...
elif condition2:
    statement2_1
    statement2_2
    ...
...
else:
    statementn_1
    statementn_2
    ...
```

Figure: General form of if-else

```
a = -5

if a > 0:
    print('a is positve.')
elif a < 0:
    print('a is negetive.')
else:
    print('a is zero.')
```

✓ 0.0s

```
a is negetive.
```

Figure: An example

Loop Statement

To execute repeated instructions multiple times:

- In Python there are two kinds of loops for repetitive tasks: **while** and **for**
- While loop is used to execute a block of statements repeatedly until a given condition is satisfied
- For loop is generally used for sequential traversal

Loop Statement

While Loop:

- When the termination condition is specified and clear, **while** can be appropriate
- For example, if you are to find out all the square numbers below 100:

```
i = 1
while i*i < 100:
    print('Square of',i,'is',i*i)
    i = i + 1

print('All the square numbers below 100 is founded')
```

✓ 0.0s

```
Square of 1 is 1
Square of 2 is 4
Square of 3 is 9
Square of 4 is 16
Square of 5 is 25
Square of 6 is 36
Square of 7 is 49
Square of 8 is 64
Square of 9 is 81
All the square numbers below 100 is founded
```

Loop Statement

For Loop:

- When you want to iterate through a sequence and find elements that fit the requirements, **For** Loop will be suitable

- At each iteration, the variable i refers to another value from the list in order

```
a = [1,2,-3,4,5]
for i in a: #go through a
    if i < 0:
        print(-i)
    else:
        print(i)
```

✓ 0.0s

```
1
2
3
4
5
```

Exception handling

- Sometimes, you will find that you will need to handle for exceptions
- The exceptions could be invalid operations (e.g., divide by 0 or compute the square root of a negative number) or a class of values that you simply are not interested in
- To handle these exceptions, you can use
  - Break
  - Continue
  - Try-except

Break

- Break statement is only allowed inside a loop body
- When break executes, the loop terminates
- In practical usage, a break statement is usually used with if statement to break a loop conditionally

Break

- Sometimes you don't have to iterate all the elements in a sequence
- Breaking the loop, when the targeted element is found

```
list = [5,25,-4,19,-23,98]
for i in list:
    if i < 0:
        break
print('The first negative number in list is:',i)
```

✓ 0.0s

```
The first negative number in list is: -4
```

Continue

- Continue statement is only allowed inside a loop body
- When continue executes, the current iteration of the loop terminates, and the execution continues with next iteration of the loop
- In practical usage, a continue statement is usually used with if statement to executes conditionally

Continue

- Stopping current iteration and continuing to the next one (do not break the loop)

```
from math import sqrt #to calculate square root
list = [5,25,-4,19,-23,98]
for i in list:
    if i < 0:
        continue
    print('Square root of',str(i),'is',str(sqrt(i)))
```

✓ 0.0s

```
Square root of 5 is 2.23606797749979
Square root of 25 is 5.0
Square root of 19 is 4.358898943540674
Square root of 98 is 9.899494936611665
```

Try-except

- Unhandled exceptions will cause your program to crash. You can handle them using try and except statement
- If the code in the try block works, the except block is skipped
- If the code in the try block fails, it jumps to the except section
- Try-except statement is not frequently used

Try-except

```
print(0/0)
```

0.0s

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
/var/folders/3k/yjmdfcsn207dh9q3c17gz4sr0000gn/T/ipykernel_55349/2624798349.py in
----> 1 print(0/0)

ZeroDivisionError: division by zero
```

```
try:
    print(0/0) #denominator is zero
except ZeroDivisionError:
    print('can not divide by zero')
```

✓ 0.0s

```
can not divide by zero
```

Figure: An example of try-except

Function in Python

Generally, the function in Python can be divided into two categories:

- Built-in function: globally predefined function in a programming language(Python)
- Self-definied function: users define function according to actual needs flexibly

Built-in function

Some examples for built-in function:

- **print**: to output something like value or string
- **len**: to get the length of a squence
- **int**: to turn a number into an integer
- **range**: to generate a **range** type variable, which is similar to a list and is frequently used with **for**

```
for i in range(3):# start with zero
    print(i)
```

✓ 0.0s

```
0
1
2
```

Function Checking

Sometimes you need to check whether a function is available especially when your code project is large:

- Use built-in function **callable()**
- you can **import** a function/package to extend your function library

```
#import a package
#now function from math is available
import math

x = 0
print(callable(math.floor))
print(callable(x)) #x is not a function
print(math.floor(1.7)) #round down
```

✓ 0.0s

```
True
False
1
```

Self-defined function

Self-defined function is important in Python programming because it can:

- Encapsulate some statemes to make the code cleaner and less error-prone
- Adapt to special requirements
- Make the code easy for reading and reasoning

Methods for self-defined function

def

- The function name and return value must be declared when using **def** with **return**

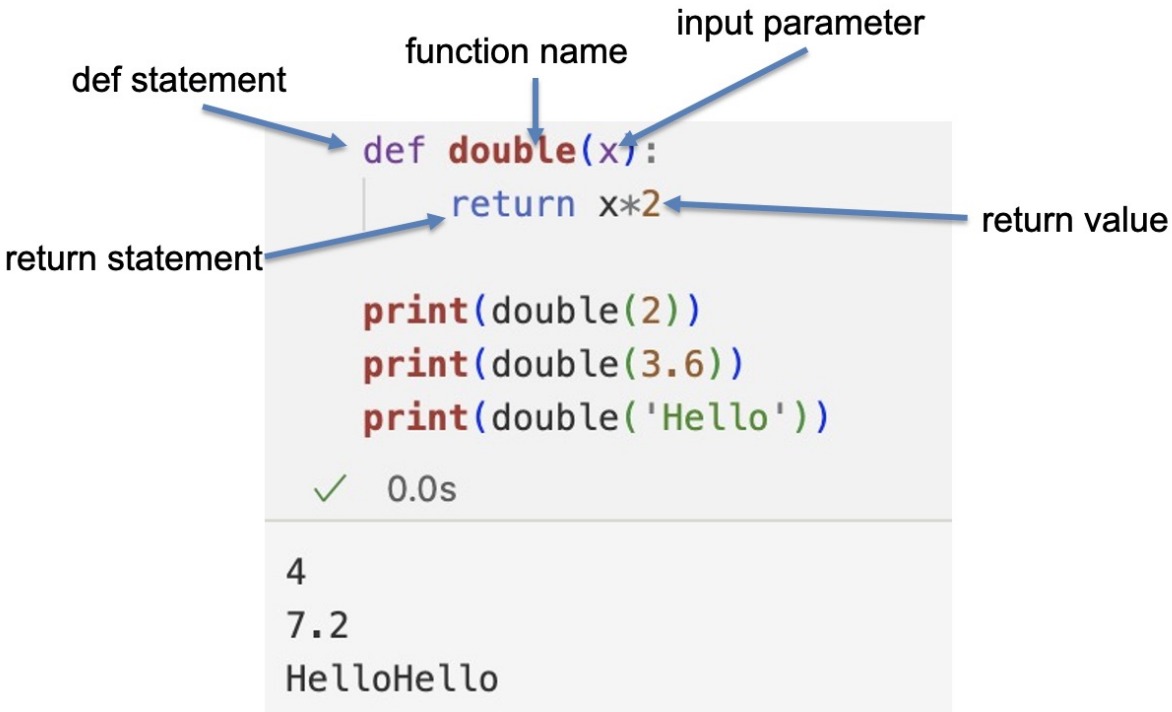
lambda

- The function name don't have to be declared but it can be used only once when using **lambda**

Methods for self-defined function

def

- The function name and return value must be declared when using **def**(generally with **return**)



Methods for self-defined function

lambda

- There is no function name using lambda
- The function can only be called from the same location

```
a = (lambda x:x+1)(3)
print(a)
```

✓ 0.0s

```
4
```

Formal and actual parameters[2]

Formal parameters

- There are usually some input parameters in a function, but these input parameters are just set to define a rule, namely formal parameters
- Formal parameters are automatically destroyed when the function completes, so they are only valid inside a function

Actual parameters

- Parameters that are actually passed to a function are called actual parameters
- The program allocates storage space for them with real values

[2] In function, the parameters sometimes are also named arguments.

Formal and actual parameters

```
def minus_one(x):# x here is formal
    return x-1

x = 2 #x here is actual
y = 1
a = minus_one(x)
b = minus_one(y)
print('a:',a)
print('b:',b)
```

✓ 0.0s

```
a: 1
b: 0
```

Figure: An example of formal and actual parameters

Modify parameters with function

- An assignment to some kinds of variables(e.g. number, tuple, string) inside a function has no effect outside

```
def change_2_one(x):
    x = 1


a = 2
change_2_one(a)
print(a)
```

✓ 0.0s

```
2
```

Modify parameters with function

- An assignment to a list inside a function has effect outside

```
def change_2_one(x):
    x[0] = 1

a = [3,2,1]
change_2_one(a)
print(a)
```

✓ 0.0s

```
[1, 2, 1]
```

Multiple parameters in function

- If the parameter name is not specified, the value is assigned in order
- When the parameter name is specified, the value is assigned by the parameter name
- Parameters whose names are not specified must come first

Multiple parameters in function

- In this case, if you specify the names of parameters, like 'age = 18' and 'name = Mike', their order can be changed

```
def name_age(name,age):
    print(name,age)

name_age('Mike',18)
name_age(18,'Mike')
name_age(age=18,name ='Mike')
name_age(name ='Mike', age=18)
```

✓ 0.0s

```
Mike 18
18 Mike
Mike 18
Mike 18
```

Figure: An example of multiple parameters

Multiple parameters in function

- However, If only some of the parameters are declared with specific names, those parameters without specific names should come first. In this case, '18' are associated with age, but 'Mike' is not, so 'Mike' should be at earlier position

```
name_age(age=18,'Mike') #Mike should come first
```

⊗ 0.0s

```
  File "/var/folders/3k/yjmdfcsn207dh9q3c17gz4sr0000gn/T/ipykernel_55349/2549555402.py", line 1
    name_age(age=18,'Mike')
                    ^
SyntaxError: positional argument follows keyword argument
```

```
name_age('Mike', age=18)
```

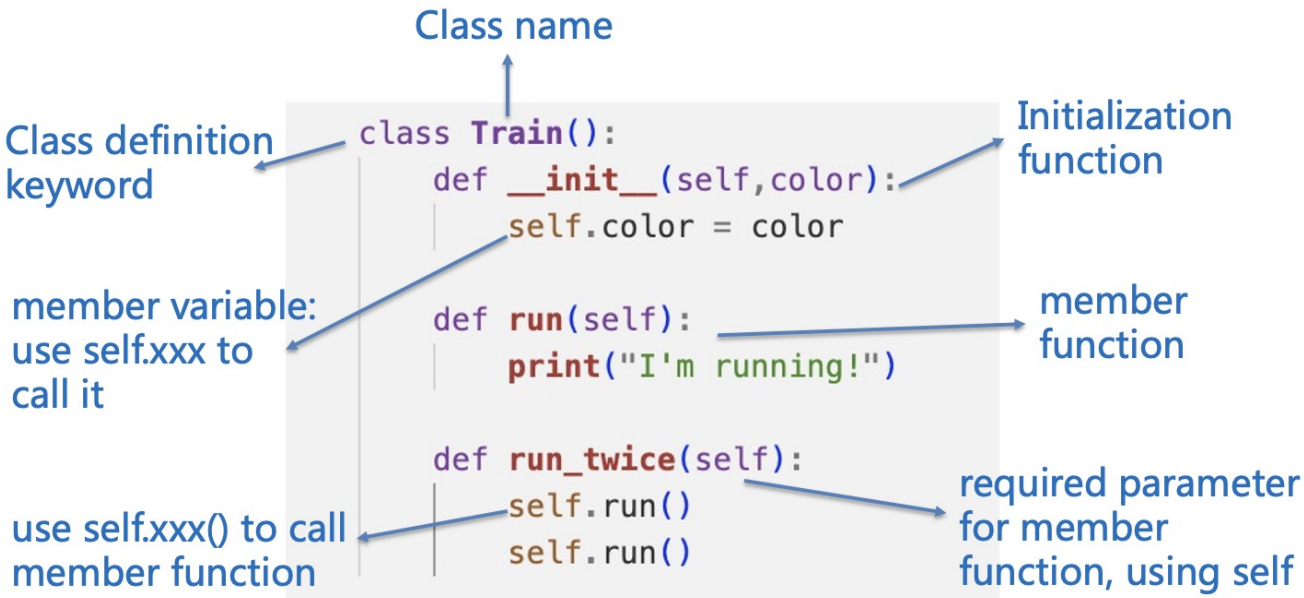
✓ 0.0s

```
Mike 18
```

Class in Python

Class

- Used to describe a collection of objects with the same properties and methods
- It is fine if you don't fully understand it for now[a]



[a] you don't have to use it now, and in future lectures related to deep learning, class may be used to build your model

Class in Python

```
class Train(): #define a class
    def __init__(self,color):
        self.color = color

    def run(self):
        print("I'm running!")

    def run_twice(self):
        self.run()
        self.run()

training = Train(color = 'red') #create a class
training.run() #call a member function in the class
print(training.color) #call a member variable
```

✓ 0.0s

```
I'm running!
red
```

Figure: An example for class

Reference Materials

Python tutorials:

- https://www.w3schools.com/python/default.asp
- https://www.tutorialspoint.com/python/index.htm

Python books:

- Andreas C. Müller, Sarah Guido, 2016. *Introduction to Machine Learning with Python: A Guide for Data Scientists*. O'Reilly Media.