Python Basics

First Edition, 2021

Programming

P. Lee

https://pyongwonlee.com/

Contents

1. Python Programming Structure

Python is a general programming language that can be used to develop many different applications such as web, gaming, and machine learning.

Modules: Python code can be placed in separate files, called modules.

- Functions and classes in a different module need to be imported before using them.
- The name of the module can be accessed using the special built-in variable: __name__
- The starting module has a special name: __main__

```
import math

print(math.__name__) # math module's name
print(__name__)      # current module's name

if (__name__ == '__main__'):
  print('You are in a main module.')
```

Blocks of Code: You can specify the block of code using the indentation.

- "*if*", "*for in*", or "*while*" statement has its own block that specifies the boundary.
- A function or a class also has its own boundary.

```
def isPositiveNumber(number):
  if number > 0:
    return True
  else:
    return False

print(isPositiveNumber(10))
print(isPositiveNumber(-10))
```

<Note> Change the indentation of the code and see how the code works.

Names and Variables

- A variable is a name of a container that has a value.
- Variable names should
  - start with an alphabet letter or underscore
  - contain only alpha-numeric (a-z,A-Z,0-9) and underscore (_)
  - not a keyword such as if, while, for, etc.
- Names are case-sensitive
- You can create a variable at any time but need to be created before using it.
  - A variable is created when a value is assigned to it for the first time.
- You do not need to specify the type of a variable. A variable can point to any type of data.
- Use the "***type()***" function to check the data type that a variable points to.

```
a = 10
print(a, type(a))
a = 'Hello'  # 10 cannot be accessed any more
A = 'World'  # a and A are different
print(a, type(a))
print(A, type(A))
```

2. Python Data Types

Python has many built-in (native) data types:

- **Numbers**: integers(1,2,3 …), floating-point numbers (1.1, 1.2, 1.3 …), complex numbers (1 + 2j)
- **Strings**: a text, a sequence of characters (letters)
- **Lists**: a sequence of values or objects
- **Tuples**: a sequence of values but cannot be modified once created
- **Dictionaries**: a sequence of key/value pairs

Some other native types:

- **Sets**: a sequence of values; do not allow duplicate values
- **Byte Arrays**: used for images and videos

Strings

- Strings are surrounded by single quotes or double quotes.
- Triple Singe Quote => multi-line string
- The '\' is used to escape special characters such as a new line '\n' or a tab '\t'.

```
greeting = 'Hello ' + "World" # String Concatenation
print(greeting)

greeting1 = 'Hello\nWorld\nGood morning'
print(greeting1)

greeting2 = '''
Hello
World
Good morning
'''
print(greeting2)
```

String as a sequence of characters or letters

- String can be thought as a list of characters.

```
greeting = 'Hello, World'
print(greeting, len(greeting), type(greeting))

for letter in greeting:
  print(letter)
```

Escape Characters

- Insert some characters as the part of a string can be tricky. The trick is to use the backslash '\'.

```
text = 'I\'m hungry.\n'
text += "\"Me too.\""
print(text)
```

Modifying strings with methods

- A string is a class and has many methods
- In general, string methods do not modify the underlying string itself. The methods return a new modified string. - -Be careful!!

```
greeting = 'HeLLo world, good MORNING!'
print(greeting.lower())
print(greeting.upper())
print(greeting.capitalize())
print(greeting) # Not changed
```

Common String Methods

- Remove spaces from the beginning or the end
    - string.strip() => returns so-called trimmed sting

- Split and Joining Strings
  - string.split() => list of strings split by space
  - string.split(separator) => list of strings split by the separator
  - separator.join(list of strings) => combine a list of strings to a single string

```
numbers = '1 2 3 4 5'
print(numbers.split()) # 5 items -- by space

fruits = 'apple,pear,mango'
fruitList = fruits.split(',') # ['apple', 'pear', 'mango'] -- by comma
print(fruitList, len(fruitList))
print(fruitList[1])
print('-'.join(fruitList)) # join with -
```

(Example) Joining the list of data by a new line and print them

```
def getInfo(name, age):
  return f'{name} is {age} years old.'

people = []
people.append(getInfo('A', 10))
people.append(getInfo('B', 20))
people.append(getInfo('C', 30))
print('\n'.join(people))
```

(Example) Splitting text line by line and stripping starting/ending spaces

```
fruits = ' apple  \n  pear \n   banana  '
fruitList = fruits.split('\n')
for fruit in fruitList:
  print(fruit)
  print(fruit.strip())
```

- Finding the text
  - string.find(): searches the string for a specified value and returns the position (0-based index)

- Replacing the text
  - string.replace()

```
greeting = 'Hello, world'
print(greeting.find('world'))   # 7
print(greeting.find('World'))   # -1 -- not found
print(greeting.lower().find('world'))   # 7

print(greeting.replace('world', 'earth')) # Hello, earth
```

String Formatting and String Interpolation

In the old version of Python, the string.format() method is used to replace the values inside the string.

- Use curly braces with the index {0}, {1} … to specify the placeholder
    - Each variable matches with the placeholder by the index
    - {0} => the first argument of the format() method
    - {1} => the second argument of the format() method

```
fruit1 = 'apple'
price1 = 10
fruit2 = 'banana'
price2 = 20

# match by the sequence (index)
message1 = 'The price of {0} is {1} dollars.'.format(fruit1, price1)
message2 = 'The price of {0} is {1} dollars.'.format(fruit2, price2)
print(message1)
print(message2)
```

[Note] Do not use this syntax if you use Python 3.6 or later. This is the reference only.

In the newer version of Python (3.6 and later), you can use the prefix '**f**' before the spring and provide values or variables directly. It is called "string interpolation".

- String interpolation is the preferred way.

```
fruit1 = 'apple'
price1 = 10
fruit2 = 'banana'
price2 = 20

# string interpolation
message1 = f'The price of {fruit1} is {price1} dollars.'
message2 = f'The price of {fruit2} is {price2} dollars.'
print(message1)
print(message2)
```

Numbers

Numbers can be an integer, a floating-number, or a complex number.

- If necessary, the numbers become “float” from “int”. (automatic conversion)

```
i = 10
f = 3.5
a = 10/3
com = 3 + 5j
print(i, type(i))
print(f, type(f))
print(a, type(a))
print(com, type(com))
```

- Formatting numbers

```
a = 12003.34564
print(a)
print(f'{a:,}')  # thousand separator
print(f'{a:.2f}') # 2 decimal points
print(f'{a:,.2f}')
print(f'You have ${a:,.2f}')
```

Boolean Values

- Boolean values represent 2 values: True or False
- It is used in branching statements such as "if"

```
t = True
f = False
print(t, f, type(t), type(False))

print(10 > 9)
print(10 == 9)
print(10 != 9)
print(10 < 9)
```

[Example] Boolean value with if statement

```
def getInterestRate(amount):
  if amount >= 1000:
    return 0.02
  else:
    return 0.01

amounts = [500, 5000]
for amount in amounts:
  interestAmount = amount * getInterestRate(amount)
  print(interestAmount)
```

Type conversion

Each type can be converted into another type if the context makes sense. The process is also called "casting".

- Conversion functions: int(), str(), bool()
- Converting to Boolean value is not intuitive. Only empty string or 0 is False.

```
i = int("10")
s = str(10.3)
print(i, type(i))
print(s, type(s))

print(bool('True'), bool('False'), bool(''), bool('Hello'))
print(bool(1), bool(0), bool(-1))
```

[Example] Splitting a string value to a string array => convert each string value to a number and get a total.

```
# sum from 1 to 10

numberText = '1,2,3,4,5,6,7,8,9,10'
numberList = numberText.split(',')
total = 0

for number in numberList:
  total = total + int(number) # type conversion is required

print(total) # 55

print(type(numberText)) # str
print(type(numberList)) # list
```

3. Operators

Arithmetic Operators -- the result is a numeric value

- + (add) - (subtraction)
- * (multiplication) / (division) // (floor division) % (modulus)
- ** (exponential)

```
a = 5
b = 3
print(a + b, a - b, a * b)
print(a / b, a // b, a % b)
print(a ** b)  # 5 * 5 * 5
```

Comparison Operators -- the result is a Boolean value

- == (equal) != (not equal)
- < <= > >=

```
a = 5
b = 3
print(a == b, a != b, a > b, a < b)
```

Logical Operators

- Logical operators work with Boolean values and the result is also a Boolean value.
- and or not

```
print(True and True, True and False, False and False)
print(True or True, True or False, False or False)
print(not True, not False)
```

Assignment Operators

- Assign the value (right side) to the variable (left side)
- A variable points to the value after the assignment
- = += -= *= /= %= **=

```
a = 10
a += 20
print(a)
a = 40  # variable points to a different value
print(a)
```

Multi assignment

- Python provides some special syntax to assign values to multiple variables in a single line.

```
# assign 1 value to multiple variables
# -- not useful much
a = b = c = "Hello"
print(a, b, c)

# assign multiple values to multiple variables
# -- multiple lines becomes 1 line
a, b, c = "Apple", "Pear", "Banana"
print(a, b, c)

# assign list items to multiple variable
# -- useful in some cases
a, b, c = ["Banana", "Pear", "Apple"]
print(a, b, c)
```

Operator Precedence

- () - parenthesis
- * / % - multiply, division
- + - - add, subtract
- == != < <= > >= - comparison operators
- and - logical AND
- or - logical OR
- = - assignment

```
print(1 + 2 * 3)  # 7
print(1 + 2 / 2 * 2)    # 3

# False or True and True ->>> Fakse and True ->>>> True
print(1 == 2 or 3 == 3 and 4 > 1)

# 9 > 10 or 1 == 1 and 1 == 1 ->>> True
x = 3 + 2 * 3  > 10 or 4 - 2 * 1.5 == 1 and 1 == 10 % 3
print(x)
```

Membership Operators -> The result is a Boolean value.

- in not in

```
print(3 in [1,2,3,4,5]) # True
print('a' not in ['a','b','c']) # False
```

4. Lists

Lists are used to store multiple values in a single name (variable).

- A list is an object and has many methods.
- A list is created by using the square brackets [ ] and separating values by a comma.
- A list can have duplicate values.
- Use the "len()" function to get the number of items in the list.

Each item in a list is indexed, which means you can access each item with an index.

- The first item has an index of [0], not [1].
    - Therefore the index of the last item is the length of a list minus 1.

```
fruits = ['mango', 'apple', 'pear', 'apple', 'banana']
print(fruits, len(fruits), type(fruits))
print(fruits[0], fruits[len(fruits)-1]) # first, last
```

A list can contain different data types.

- But it is not a recommended practice.

```
fruits = ['mango', 2, 'banana', 1]
print(f'{fruits[0]} is {fruits[1]} dollars.')
print(f'{fruits[2]} is {fruits[3]} dollars.')
```

A list is changeable. You can add, remove, and update the items at any time.

- You can create a list without any items. – Just do not specify any item inside [].

```
fruits = ['mango', 'banana']
fruits[1] = 'apple'  # you can update each item directly
print(fruits)
```

```
# add items
fruits = []  # it creates a list object without any items in it
fruits.append('apple') # add at the end
fruits.append('mango') # add at the end
print(fruits)
fruits.insert(1, 'pear') # add at the index 1 and push others
print(fruits)
```

Removing items from a list

- It is easy to remove numbers and strings with the "remove()" method.
- Another way to delete an item is to use the del keyword with an index.

```
# remove items
fruits = ['mango', 'banana', 'apple', 'pear', 'apple']
fruits.remove('apple')  # first match
print(fruits)
fruits.remove('apple')
print(fruits)
```

```
# delete items
fruits = ['mango', 'banana', 'apple', 'pear', 'apple']
del fruits[2]
print(fruits)
del fruits[len(fruits)-1]  # last
print(fruits)

# delete all items -- clear -- the list becomes empty
fruits = ['mango', 'banana', 'apple', 'pear', 'apple']
fruits.clear()
print(fruits)
```

Looping through a list

- The for loop does not require an index.

```
fruits = ['mango', 'banana', 'apple', 'pear', 'apple']

# without an index
for f in fruits:
  print(f)

# with an index - use range() function
for index in range(len(fruits)):
  print(index, fruits[index])
```

Sorting a list

- list.sort(reverse, key)
  - *reverse:* True or False (default)
  - *key:* a function to specify the sorting criteria - advanced feature (ignore now)
- sort() updates the current list itself.

```
fruits = ['mango', 'banana', 'apple', 'Melon', 'pear', 'Apple']
fruits.sort() # Uppercase comes before lower case
print(fruits)

numbers = [10, 3, -5, 0, 22, 15]
numbers.sort(reverse=True)
print(numbers)

# reverse() function is different
numbers = [10, 3, -5, 0, 22, 15]
numbers.reverse()
print(numbers)
```

[Example] range() function

- The range() function creates a sequence of numbers with a start value, a stop value (not included), and a step value.

```
# stop only
r = range(10)
print(r, type(r))

# start, stop
r = range(1, 10)
print(r, type(r))

# start, stop, step
r = range(1, 21, 4)
print(r, type(r))
```

The return value is not a list object. -- It is a range object.

- You can convert a range object to a list object.
    - This technique is commonly used to get a list of numbers.

```
r = range(10)
l = list(r)
print(r, type(r), l, type(l)) # range and list

l.append(10)
print(l)
r.append(10)  # this will fail
```

```
# odd numbers from 1 to 100
odd_number_range = range(1, 100, 2)

odd_numbers = list(odd_number_range)
print(odd_numbers)
```

The range object can be used with the for loop.

- It is a common practice to loop through a list with an index using the range object.

```
names = ['A', 'B', 'C', 'D', 'E']
for index in range(len(names)):
  print(index, names[index])
```

[Advanced Example] Remove the items from a list for an object & Object Equality

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

people = [Person('A'), Person('B'), Person('C')]
print(people)
people.remove(Person('B'))  # what will happen here???
print(people)
```

Even though the names of Person objects are the same, the code will fail because they are actually different objects. To remove the object, you can use an index or you need to use the same object from the list.

```
class Person:
  def __init__(self, name):
    self.name = name
  def __repr__(self):
    return self.name

people = [Person('A'), Person('B'), Person('C')]
print(people)
people.remove(people[1]) # remove the second item
print(people)
del people[0] # remove the first item
print(people)
```

But there is another problem. You need to know the index. You only know the name of the person is 'B' but do not know where this person is located in a list.

The main problem is to understand how objects are equal.

- Numbers and strings are used to compare values to check the equality.
- Objects are equal when they are exactly the same object – points to the same memory address.

```
print(Person('A') == Person('A'))  # False
```

```
b = Person('B')
people = [Person('A'), b, Person('C')]
print(b == people[1]) # True
```

You can modify the default behavior by overriding the __eq__ method in your custom class.

```
class Person:
  def __init__(self, name):
    self.name = name
  def __eq__(self, other):
    return self.name == other.name

print(Person('A') == Person('A'))  # True
```

Here is the final code to delete the object in a list.

```
class Person:
  def __init__(self, name):
    self.name = name
  def __repr__(self):
    return self.name
  def __eq__(self, other):
    return self.name == other.name

people = [Person('A'), Person('B'), Person('C')]
print(people)
people.remove(Person('B'))
print(people)
```

5. Tuples

Tuple is similar to List but cannot be changed after it is created.

- It uses parentheses and comas (, , ,) and allows duplicates.
- A tuple can contain different data type values.

```
# list vs. tuple
fruitList = ['apple', 'pear']
fruitTuple = ('apple', 'pear')

print(fruitList, type(fruitList))
print(fruitTuple, type(fruitTuple))
```

You can access a tuple with an index just like a list.

```
fruits = ('mango', 'apple', 'pear', 'apple', 'banana')

print(fruits, len(fruits), type(fruits))
print(fruits[0], fruits[len(fruits)-1]) # first, last
```

List vs. Tuple

- *List*: mutable, variable length
- *Tuple*: non-mutable, fixed length

- A tuple is faster and safer to work with -- if you know items that are not changed in advance.

```
fruits = ('mango', 'apple', 'pear')
fruits[2] = 'banana'  # error
```

Looping through a tuple

```
fruits = ('mango', 'apple', 'pear')
for f in fruits:
  print(f)
```

Accessing the range of indexes in tuples and lists:

- The end index is not included.

```
fruits = ('mango', 'apple', 'pear', 'apple', 'banana')
print(fruits[1:3], fruits[2:], fruits[:4])

fruits = ['mango', 'apple', 'pear', 'apple', 'banana']
print(fruits[1:3], fruits[2:], fruits[:4])
```

6. Dictionaries

A dictionary is a sequence of Key/Value pairs.

- A dictionary is created by using curly braces { }.
- Items are separated by a comma.
- A key and a value are separated by a colon.

- Accessing an item in a dictionary – Use the key.

```
fruits = { 'apple': 1, 'pear': 2, 'banana': 0.5 }
print(fruits, len(fruits), type(fruits))
print(fruits['apple'], fruits['banana'])
```

Adding, Editing, and Removing items in a dictionary

```
fruits = { 'apple': 1, 'pear': 2 }
fruits['banana'] = 0.5  # update the value directly with a key
print(fruits)
fruits['pear'] = 1.5
print(fruits)
```

```
fruits = { 'apple': 1, 'pear': 2, 'banana': 0.5 }
del fruits['pear'] # remove the item directly
print(fruits)
```

```
fruits = { 'apple': 1, 'pear': 2, 'banana': 0.5 }
fruits.clear()  # remove all items
print(fruits)
```

Looping through a dictionary

```
fruits = { 'apple': 1, 'pear': 2, 'banana': 0.5 }
for f in fruits:
  print(f) # key only
  print(f'{f} is {fruits[f]} dollars')  # key and value
```

Getting keys as a list -- keys()

```
fruits = { 'apple': 1, 'pear': 2, 'banana': 0.5 }

keys = list(fruits.keys())
print(keys[0])
print(keys[1])

for key in keys:
  print(key, fruits[key])
```

Getting values as a list -- values()

```
fruits = { 'apple': 1, 'pear': 2, 'banana': 0.5 }

values = list(fruits.values())
print(values[0])
print(values[1])

for value in values:
  print(value)
```

7. Lists, Dictionaries – Advanced Features

Matrix - Nested Lists

(Example) 2 dimensional matrix

1	2	3
4	5	6
7	8	9

```
my_3x3_matrix = [[1,2,3],[4,5,6],[7,8,9]]

print(my_3x3_matrix)
print(my_3x3_matrix[0][0])
print(my_3x3_matrix[0][1])
print(my_3x3_matrix[1][2])

row_count = len(my_3x3_matrix)
print(row_count)

for row in my_3x3_matrix:
  column_count = len(row)
  print(column_count, row)
```

List Comprehensions

You can transform a list to another list with for loop.

```
fruits = ['apple', 'pear', 'mango']
fruits_upper = []

for f in fruits:
  fruits_upper.append(f.upper())
print(fruits_upper)
```

The *comprehension* syntax can do the same job in a single line.

```
fruits = ['apple', 'pear', 'mango']

# comprehenstion ===>  [new_item for old_item in list]
fruits_upper = [f.upper() for f in fruits]

print (fruits_upper)
```

Comprehension can be used for filtering.

- Creating a smaller list from an original list with the items that match with the condition.

```
fruits = ['apple', 'pear', 'mango', 'orange']
fruits_1 = []

for f in fruits:
  if f in ['apple', 'orange']:
    fruits_1.append(f)

print(fruits_1)  # only apple and orange
```

The code can be done using the comprehension.

```
fruits = ['apple', 'pear', 'mango', 'orange']

# comprehension with a condition
#  ===>  [new_item for old_item in list if condition]
fruits_1 = [f for f in fruits if f in ['apple', 'orange']]
print(fruits_1)  # only apple and orange
```

(Example -Advanced) String Join and List Comprehensions

```
class Product:
  def __init__(self, name):
    self.name = name
  def getProductInfo(self):
    return f'The product name is {self.name}.'

class ProductList:
  def __init__(self, products):
    self.products = products

  def getProductList1(self):
    result = ''
    for product in products:
      result += product.getProductInfo() + '\n'
    return result

  # this can be done simpler
  def getProductList2(self):
    return '\n'.join([product.getProductInfo() for product in self.products])

products = [Product('Bread'), Product('Milk'), Product('Meat')]
productList = ProductList(products)

print(productList.getProductList1())
print('----------------------')
print(productList.getProductList2())
```

8. Control Flow

Conditions and Branching

(Review) in, not in

- Can be used with a list, a tuple, or a dictionary
- For dictionary, a key is checked.

```
print (1 in [1, 2, 3])
print (2 not in (1, 2, 3))
print ('apple' in { 'apple': 1, 'pear': 2})
print ('pear' not in { 'apple': 1, 'pear': 2})
```

if elif else

- Branching based on the condition
- Executes the block of code when the condition is evaluated as True

```
import datetime
currentTime = datetime.datetime.now().hour

if currentTime < 12:
  print(currentTime, 'Good morning')
else:
  print(currentTime, 'Good afternoon')
```

```
def getGrade(score):
  grade = ''
  if (score >= 90):
    grade = 'A'
  elif (score >= 80):
    grade = 'B'
  elif (score >= 70):
    grade = 'B'
  elif (score >= 60):
    grade = 'B'
```

```
  else:
    grade = 'F'
  return grade

scores = [89, 77, 56, 65, 95]
for score in scores:
  print(score, getGrade(score))
```

Ternary Operators, or Conditional Expressions

- There is a shorthand if else statement.

```
a, b, c = 5, 10, 0 # multiple value assignment

# normal if else
if a > b:
  max = a
else:
  max = b
print(max)

# short-hand if else
max = a if a > b else b
print(max)
```

- The ternary if else statement does not need to be an assignment.

```
import datetime as dt

print('Good morning') if dt.datetime.now().hour < 12 else print('Good afternoon')
```

while loop

- The while loop repeats the block of code while the condition is true.

```
number = 1
total = 0
while number <= 10:
  total += number
  number += 1

print(number, total)
```

- continue: stops the current iteration (ignores all the next code in the block) , and continues with the next iteration
- break: stop the loop immediately and exit the block, no more iteration

```
# counting odd numbers -- to 10 odd numbers
countOfOddNumbers = 0
number = 0

while True:
  number += 1
  if number % 2 == 0:
    continue
  countOfOddNumbers += 1
  print(f'Inside the loop: {number}, {countOfOddNumbers}')
  if countOfOddNumbers == 10:
    break

print(f'Outside the loop: {number}, {countOfOddNumbers}')
```

9. Functions

A function is **a block of code** that can be reused without repeating it.

- The def keyword is used to start the definition of a function.
- The name of the function.
- (Parameter lists)
- A block of code to run when the function is called.

Parameters and **Arguments**

- *Parameters*: the variable names for the input for a function. They are just variable names that can be used inside of a function and point to the data.
- *Arguments*: the real data that are passed to a function when the function is called or executed.

```
# function definition
# name: say_hello
# parameter: name --> you can use this variable inside the function
# A parameter points to the value passed from the function call (argument)
def say_hello(name):
  print(f'Hello, {name}')

names = ['Homer', 'Bart']

for name in names:
  # call the function here
  # name points to 'Homer' or 'Bart' for each iteration
  # the data is passed to a function
  # name - 'Homer' and 'Bart' is an argument
  say_hello(name)
```

A function can return a single value back to the caller using the return statement.

- When you call (execute) a function, you need to assign a return value to a variable.

```
# a, b ==> parameters
# returns the sum of 2 values back to the caller
def add(a, b):
  return a + b

# 10, 20 ==> arguments
# result is a variable to point to the return value of a function
result = add(10, 20)
print(result, type(result))
```

[Problem 1] Do not return a value when it is required.

- If there is no return value, the function returns None.

```
# Problem 1 -- forget to return a value
def add(a, b):
  c = a + b

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

[Problem 2] Return statement finishes a function

- When a function reaches the return statement, the function returns a value to a caller and exits the function. Any code after the return statement is ignored.

```
def add(a, b):
  return 'This is add function.'
  c = a + b  # this is ignored
  return c   # this is ignored

result = add(10, 20)

print(result, type(result))
```

Using **keyword arguments**

- Send arguments with the parameter name = argument value syntax.
- The order of the arguments does not matter.
- Positional arguments must be placed before keyword arguments.

```
def product(name, price):
  print(f'{name} is {price} dollars')

product('Bread', 4)
product('Milk', price=5)

# the order does not matter
product(price=1, name='Apple')
product(name='Apple', price=1)

# but keyword arguments cannot be placed before the positional ones
# product(name = 'Eggs', 6) # Error
```

Default values

- In general, the number of arguments should match the number of parameters.
- By specifying the default value in the function definition, the argument becomes optional.

```
def product(name, price=1):
  print(f'{name} is {price} dollars')

product('Bread')  # the valur for the price is not passed. 1 is used.
product('Milk', 5)
```

Function Recursion

- You can call the function inside of its function.
- Inside of a function, there is a condition to end the recursive function calls. Be careful not to call the function indefinitely.

(Example) Fibonacci Sequence: 1,1,2,3,5,8,13 …

f(0) = 0
f(1) = 1
f(2) = f(0) + f(1) = 2
f(3) = f(1) + f(2) = 3
f(4) = f(2) + f(3) = 5

```
def fibonacci(n):
  if n <= 0:
    return 0
  elif n == 1:
    return 1
  else:
    return fibonacci(n-2) + fibonacci(n-1)

for n in range(11):
  print(f'{n} - {fibonacci(n)}')
```

10. Modules

A module is a separate code that is located in a file for reuse.

- A Python module is saved in a file with the file extension .py

```
# mymodule.py

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

PI = 3.14
```

```
# in the main file

import mymodule

print(mymodule.add(1,2))
print(mymodule.PI)
```

Import modules

To use a module in a different module, you need to import the module before using it.

- Once a module is imported, you can use classes, functions, and variables in the module.
- There are a couple of ways to import a module.

```
# import + module name
import math

# You can access the content of the module only through a module name
print (math.pi)
print (math.sin(math.pi))
```

```
# import + module name + as + new name
import math as m

# You can the module using the new name
print (m.pi)
# print (math.sin(math.pi)) # this does not work anymore
```

```
# from + module name + import + functions/classes/variables
from math import pi, cos

# You can only access the ones you specified
# You do not need to specify the module name. It is an error actually.
print (cos(pi))

# print(math.pi) # using a module name is an error
# print (sin(2, 3)) # this does not work - sin is not imported
```

```
# new names for the functions, classes, variables
from math import pi as phi, cos as c, sin

print (phi) # try pi, -- error
print (c(phi), sin(phi))
```

In some cases, you can import everything from a module so that you do not need to specify the module name.

```
from math import *

# check the difference: round, floor, ceil, and trunc functions
print (round(1.4), round(1.5), round(-1.4), round(-1.5))
print (floor(1.4), floor(1.5), floor(-1.4), floor(-1.5))
print (ceil(1.4), ceil(1.5), ceil(-1.4), ceil(-1.5))
print (trunc(1.4), trunc(1.5), trunc(-1.4), trunc(-1.5))

print(hypot(3, 4)) # hypotenuse

print(radians(45), radians(90), radians(180)) # 180 degree = pi radian

print(sin(radians(0)), sin(radians(30)), sin(radians(90)))
print(sin(0), sin(pi/6), sin(pi/2))
print(cos(0), cos(pi/6), cos(pi/2))
```

Checking module name:

- Use the built-in variable: __name__
- The starting module has a name: __main__

```
# mymodule.py

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

```
# in the main file

import mymodule as my_math

print(__name__)
print(my_math.add(1,2))
```

Python provides many built-in modules.

- The dir(…) function can be used to show the entities (variable, functions, and classes) of the module or a class.

```
import platform
import datetime as dt

print(dir(platform))  # platform module
print(dir(dt)) # datetime module
print(dir(dt.datetime)) #  datetime.datetime class
```

```
import platform
import datetime as dt

print(platform.system())
print(platform.machine())
print(platform.processor())
print(platform.python_version())

print(dt.datetime.now())
```

11. OOP (Object Oriented Programming)

Class & Object

A class defines the common attributes and behaviors shared by its objects.

- Methods: the same behaviors
- Properties: the same attributes

An object is an *instance* of a class.

- An object can be uniquely identified by its name, and it defines a state which is represented by the values of its properties at a particular time.

Generally, the name of a class starts with an upper case, and a variable name of an object starts with a lower case.

Checking the Entities in a Class

The dir(..) function shows all entities (variables and methods).

```
class Person:
  pass  # empty class

print(dir(str)) # str class
print()
print(dir(Person)) # custom class
```

[Note] Even an empty class has many members already. Python constructs the basic structure for you.

Any members that start with the double underscore (__) are used only inside of a class code. You cannot access these members from outside using an object variable.

Define a class

- Constructor: __init__ (self, ...)
  - __init__() : 'double underscore'
  - always automatically executed when the class is being initiated (when an object is created)
- Inside the __init__, define properties that can be used inside of a class.

self

- The self parameter is used to access properties and methods inside of a class.
- The self parameter must be the first parameter of class methods.
  - When a method is called, the caller does not send the argument for the self parameter. It is automatically assigned by a Python runtime.

Using Properties

- Properties are the data inside of a class.
  - Properties are defined in the __init__ method.
- You can read and write properties directly from inside of a class or through an object.
- You need to use the self parameter to access properties inside the class.

```
# module

class Car:
  def __init__(self, model, color):
    self.model = model
    self.color = color

  def getInfo(self):
    return f'{self.model} - {self.color}'

  def changeColor(self, newColor):
    self.color = newColor
```

```
# main
from mymodule import Car

car = Car('Civic', 'Red')
print(car.getInfo())
car.changeColor('Yello')
print(car.getInfo())
```

You need to use the object variable to access properties outside the class.

- It is possible to access the properties outside of a class definition.
- But in general, it is not a good idea to access properties directly from outside of a class.

```
# module

class Car:
  def __init__(self, model, color):
    self.model = model
    self.color = color
```

```
# main
from mymodule import Car

car = Car('Civic', 'Red')
print(f'{car.model} - {car.color}')  # access propeties of an object
car.color = 'Yello'  # you can even update the value
print(f'{car.model} - {car.color}')
```

Using Methods

Functions defined in a class do actions.

- The first parameter of a method must be self.
    - When you call the method, the caller does not send an argument for the self parameter. Python run-time sends an object reference as an argument automatically.
- A method can access properties and other methods of a class through the self parameter.

```
# module

class Car:
  def __init__(self, color, model):
    self.color = color
    self.model = model
    self.maxSpeed = 100  # the value does not need to come from outside

  def print(self):
    print(f'My {self.color} {self.model} can run up to {self.maxSpeed} kms/h.')
```

```
# main
from mymodule import Car

myCar = Car(color='white', model='Civic')
myCar.print()
```

[Example] Bank Account

```
# module

class BankAccount:
  def __init__(self):
    self.amount = 0
    self.interestRate = 0.01

  def deposit(self, amount):
    # update the property directly
    # self.amount ---> object property
    # amount ---> passed from outside when the method is called
    self.amount += amount

  def withdraw(self, amount):
    self.amount -= amount

  def getBalance(self):
    return self.amount * (1 + self.interestRate)
```

```
# main

from mymodule import BankAccount

acc = BankAccount()
print(type(acc))
print(f'My balance is {acc.getBalance()}')

acc.deposit(1000)
print(f'My balance is {acc.getBalance()}')

acc.withdraw(500)
print(f'My balance is {acc.getBalance()}')
```

12. OOP - Encapsulation

The main benefit of OOP is hiding the complex implementation details inside the class code.

- The user of a class does not need to know the inner details of a class.
- A user needs to know how to create an object and which methods to call to perform the desired actions.

Encapsulation means hiding details.

- By default, you can access any properties and methods using an object variable.
- The best practice is to hide properties inside the property. In python, if the property variable name starts with an underscore '_', it is a signal that you should not access this property outside of a class code.

Let's revisit the previous Bank Account code and modify the class using the underscore property name.

```
class BankAccount:
  def __init__(self):
    self._amount = 0
    self._interestRate = 0.01

  def deposit(self, amount):
    # update the property directly
    # self._amount ---> object property
    # amount ---> passed from outside when the method is called
    self._amount += amount

  def withdraw(self, amount):
    self._amount -= amount

  def getBalance(self):
    return self._amount * (1 + self._interestRate)
```

It is still allowed to access the property directly.

```
# main
from mymodule import BankAccount

acc = BankAccount()

acc._amount = 2000  # don't do it

print(f'My balance is {acc.getBalance()}')
```

[Note] All Python developers are aware that any properties start with an underscore should not be accessed directly. It is a common practice.

13. OOP - Composition

"Has A" relationship

- Composition is a relationship between object. An object can have other objects as its properties.

[Example] A music album has many songs.

```
# module

class Song:
  def __init__(self, title):
    self._title = title

  def getSongInfo(self):
    return f'Song - {self._title}'

class MusicAlbum:
  def __init__(self, title, songs):
    # album title is different from a song title
    self._title = title
    self._songs = songs

  def getAlbumInfo(self):
    result = f'Album - {self._title}. \n'
    # check join the list and comprehension
    result += '\n'.join([song.getSongInfo() for song in self._songs])
    return result
```

```
# main

from mymodule import Song, MusicAlbum

song1 = Song('Happy Tune')
song2 = Song('Loud Loud')
song3 = Song('Piano Concerto')

songs = [song1, song2, song3]

album = MusicAlbum('My Music', songs)

print(album.getAlbumInfo())
```

Representation of an object

When you print the object, it shows the internal state of an object. It is the representation of the current state.

```
class Song:
  def __init__(self, title):
    self._title = title

song1 = Song('Best')
song2 = Song('Best')
print(song1)
print(song2)
```

The default representation of an object shows the module name, the class name and the memory address of an object.

- Notice that each object has a different memory address.
- In general, this information does not show the status of an object.

```
<__main__.Song object at 0x000002116BF30100>
<__main__.Song object at 0x000002116BF30DC0>
```

To provide the custom representation, you can override the __repr__ function in a class.

- It returns a string value to represent the current status of an object.

```
class Song:
  def __init__(self, title):
    self._title = title

  def __repr__(self):
    return f'This is a Song - {self._title}'

  def setTitle(self, newTitle):
    self._title = newTitle

song1 = Song('Best')
print(song1)

song1.setTitle('Just Loud')
print(song1)
```

14. OOP - Inheritance

"Is A" relationship

- Inheritance allows a class to inherit all the methods and properties from another class.
    - Parent class or base class
    - Child class, derived class or sub-class

Defining a child class

- The properties in a parent class need to be initialized by calling super().__init__()
- It is important to call the constructor of a parent class inside of a child.
- You can add any extra properties inside the constructor of a child class.

```
# module
class Person:
  def __init__(self, firstName, lastName):
    self._firstName = firstName
    self._lastName = lastName

class Employee(Person):
  def __init__(self, firstName, lastName, department):
    super().__init__(firstName, lastName)  # properties in the parent
    self._department = department  # add a new property for a child

  def describe(self):
    return f'{self._firstName} - {self._lastName} - {self._department}'
```

```
# main
from mymodule import Employee, Person

emp = Employee('A', 'B', 'Finance')
print(emp.describe())
```

Let's improve the previous example.

What if the Person class also has a describe() method and the Employee class wants to use it? Both a child and a parent have the method with the same name.

- The child class inherits all methods and properties.
    - The child can use the self parameter to access the parent.

- super() function
  - If a parent and a child have the method with the same name, using self inside a child only calls the method in a child.
  - Inside a child class, you can call access the methods in a parent using the super() function.

- Also, you can update the __repr__ function to provide a better description.

```
# module

class Person:
  def __init__(self, firstName, lastName):
    self._firstName = firstName
    self._lastName = lastName

  def describe(self):
    return f'{self._firstName} - {self._lastName}'

  def __repr__(self):
    return self.describe()


class Employee(Person):
  def __init__(self, firstName, lastName, department):
    super().__init__(firstName, lastName)  # properties in the parent
    self._department = department  # add a new property for a child

  def describe(self):
    return f'{super().describe()} - {self._department}'

  def getParentInfo(self):
    return super().describe()

  def getChildInfo(self):
    return self.describe()

  def __repr__(self):
    return self.describe()
```

```
# main
from mymodule import Employee, Person

emp = Employee('A', 'B', 'Finance')
print(emp.describe())
print(emp.getParentInfo())
print(emp.getChildInfo())

person = Person('C', 'D')
print(person)
print(emp)
```

15. OOP – Polymorphism

If classes have the same method name, you can call it without knowing which object you are using.

```
# module

class Dog():
  def cry(self):
    return f'Bark! Bark!'

class Tiger():
  def cry(self):
    return f'Uh Hung!'
```

```
# main
from mymodule import Dog, Tiger

animals = [Dog(), Tiger()]
for animal in animals:
  print(animal.cry())# it just calls cry() method in each object
```

Method Overriding with inheritance

- The child class can use the same method name but it will replace the parent one.

```
# mymodule

class Animal:
  def fly(self):
    return 'Not sure.'

class Bird(Animal):
  def fly(self):
    return 'I can fly.'

class Dog(Animal):
  def fly(self):
    return 'I cannot fly.'
```

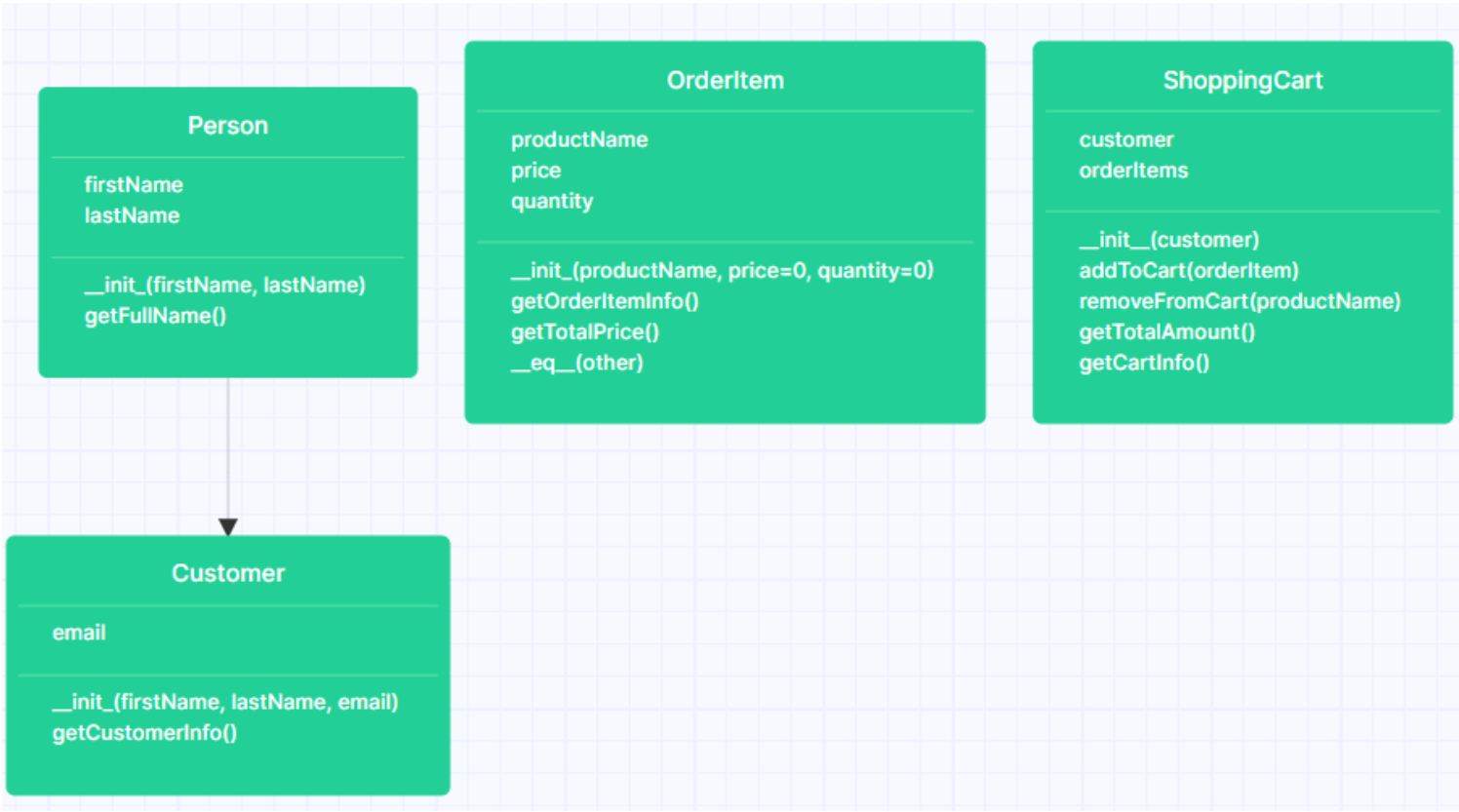
```
# main

from mymodule import Animal, Bird, Dog

animals = [Animal(), Bird(), Dog()]
for animal in animals:
  print(animal.fly())
```

16. OOP – Example

Shopping Cart Application




```python
# mymodule

class Person:
  def __init__(self, firstName, lastName):
    self._firstName = firstName
    self._lastName = lastName

  def getFullName(self):
    return f'{self._firstName} {self._lastName}'
```

```
class Customer(Person):
  def __init__(self, firstName, lastName, email):
    super().__init__(firstName, lastName)
    self._email = email

  def getCustomerInfo(self):
    return f'{self.getFullName()} - {self._email}'
```

```
class OrderItem:
  def __init__(self, productName, price=0, quantity=0):
    self._productName = productName
    self._price = price
    self._quantity = quantity

  def getOrderItemInfo(self):
    info = f'{self._productName} - $ {self._price:,.2f} per each'
    info += f', Quanity: {self._quantity}'
    info += f', Amount: $ {self.getTotalPrice():,.2f}'
    return info

  def getTotalPrice(self):
    return self._price * self._quantity

  def __eq__(self, other):
    return self._productName.lower() == other._productName.lower()
```

```
class ShoppingCart():
  def __init__(self, customer):
    self._custoemr = customer
    self._orderItems = []

  def addToCart(self, orderItem):
    self._orderItems.append(orderItem)

  def removeFromCart(self, productName):
    # using the OrderItem __eq__
    # if the product name is the same, 2 order items are treated the same
    itemToRemove = OrderItem(productName)
    self._orderItems.remove(itemToRemove)
```

```
  def getTotalAmount(self):
    total = 0
    for item in self._orderItems:
      total += item.getTotalPrice()
    return total

  def getCartInfo(self):
    info = '\n\n---------------------------------- \n'
    info += f'Customer: {self._custoemr.getCustomerInfo()} \n'
    info += '---------------------------------- \n'
    info += f'Total Price: $ {self.getTotalAmount():,.2f} \n'
    info += '\n'.join([item.getOrderItemInfo() for item in self._orderItems])
    return info
```

Here is the main module now.

```
# main
from mymodule import Customer, OrderItem, ShoppingCart

customer = Customer('John', 'Doe', 'jo@test.com')
cart = ShoppingCart(customer)

cart.addToCart(OrderItem('Bread', 3.50, 3))
cart.addToCart(OrderItem('Milk', 6.99, 1))
cart.addToCart(OrderItem('Pizza', 10.99, 1))
cart.addToCart(OrderItem('Ice Cream', 5.22, 2))
print(cart.getCartInfo())

cart.removeFromCart('Pizza')
print(cart.getCartInfo())
```

17. Sample applications with the random module

Python has a built-in module that you can use to generate random numbers.

Generating random numbers

```
import random as r

# random(): Return a random number between 0.0 and 1.0
for n in range(10): # 10 times
  print(r.random())
```

```
import random as r

# randint(start, stop): Return a random integer within a range

# start and stop values are included.
for n in range(10): # 10 times
  print(r.randint(1, 6)) # simulating dice
```

Selecting an item randomly from a list

```
import random as r

fruits = ['apple', 'pear', 'mango', 'banana']

# choice(list): Return a randomly selected item
for n in range(10): # 10 times
  print(r.choice(fruits))
```

Randomizing the order of a list

```
import random as r

numbers = list(range(10))
print(numbers)
print('-------')

# shuffle(list): this changes the original list.
# It does not return a new list!!!!
for n in range(10): # 10 times
  r.shuffle(numbers)
  print(numbers)
```

Guessing Number Game

```
import random as r

# Guessing Game

answer = r.randint(1, 100) # from 1 to 100

tryCount = 0
while True:
  tryCount += 1
  guess = int(input('Guess the number? '))
  if guess == answer:
    break
  elif guess > answer:
    print('Too big.')
  else:
    print('Too small.')

print(f'Correct! The answer is {answer} and you tried {tryCount} times.')
```

18. Python Advanced Features - Dictionaries

Dictionary is very powerful and can be used in many different ways.

Getting keys and values as a list

- The return value of keys() and values() is not a list object. You can convert it to a list.
- You can loop through keys and values easily without converting to a list object. It works the same way.

```
states = {'AK':'Alaska', 'CA':'California', 'FL':'Florida', 'NY':'New York'}

print(states.keys())
print(list(states.keys())) # convert to a list
print('-----------------------')
for key in states.keys():
  print(key)

print() # empty line
print(states.values())
print(list(states.values())) # convert to a list
print('-----------------------')
for key in states.values():
  print(key)
```

Converting a dictionary into a list of tuples

- Each item in a dictionary becomes a tuple.
    - The key becomes the first value
    - The value becomes the second value
- The return value of items() is not a list object. But you can loop through it.

```
states = {'AK':'Alaska', 'CA':'California', 'FL':'Florida', 'NY':'New York'}

print(states.items())
statesList = list(states.items())
print(statesList)
```

```
print('-----------------')
for state in states.items():
  print(state, state[0], state[1])

print('-----------------')
for state in list(states.items()):  # the same as before
  print(state, state[0], state[1])
```

Looping Through Items in a dictionary

There is another way to loop through a dictionary using items() method.

- Check another Python syntax to assign a values in a list to multiple variables.

```
fruits = ['apple', 'pear', 'mango', 'banana']

# fruits list has 4 values
# Each value is assigned to a variable in the same order
# the number of variables should be the same as the number of items
a, p, m, b = fruits

print(a, p, m, b)
```

- For a dictionary, you need to items() function to access the each item.

```
states = {'AK':'Alaska', 'CA':'California', 'FL':'Florida', 'NY':'New York'}

# Each value is assigned to a variable in the same order
a, c, f, n = states.items()

print(a, c, f, n) # items
print(type(a)) # -- each item is a tuple
print(a[0], c[0], f[0], n[0]) # keys
print(a[1], c[1], f[1], n[1]) # values
```

- Let's loop through a dictionary.
    - The loop variable only contains the key value.

```
states = {'AK':'Alaska', 'CA':'California', 'FL':'Florida', 'NY':'New York'}

for state in states:
  # state only has a key
  print(state, states[state])
```

- With *items*() function and multiple variables, you can access the key and value at the same time.

```
states = {'AK':'Alaska', 'CA':'California', 'FL':'Florida', 'NY':'New York'}

for key,value in states.items():
  print(key, value)
```

- The following short names are commonly used.

```
states = {'AK':'Alaska', 'CA':'California', 'FL':'Florida', 'NY':'New York'}

for (k,v) in states.items():
  print(k, v)
```

Dictionary Comprehension

Just like a list comprehension, you can transform one dictionary to another dictionary.

- Syntax 1: *for key in dictionary*

```
states = {'AK':'Alaska', 'CA':'California', 'FL':'Florida', 'NY':'New York'}

# dictionary comprehension syntax
# { newKey: newValue for key in dictionary }
statesOpposite = { states[key]: key for key in states }
print(statesOpposite)
```

- Syntax 2: *for (key, value) in dictionary*
    - For a dictionary comprehension, the items() is commonly used.

```
states = {'AK':'Alaska', 'CA':'California', 'FL':'Florida', 'NY':'New York'}

# dictionary comprehension syntax
# { newKey: newValue for (k,v) in dictionary.items }
statesOpposite = { v: k for (k,v) in states.items() }
print(statesOpposite)
```

```
states = {'AK':'Alaska', 'CA':'California', 'FL':'Florida', 'NY':'New York'}
statesUpperCase = { k: v.upper() for (k,v) in states.items() }
print(statesUpperCase)
```

- Syntax 3: *for (key, value) in dictionary if condition*
    - You can filter the items using the condition.

```
states = {'AK':'Alaska', 'CA':'California', 'FL':'Florida', 'NY':'New York'}
someStates = { k: v for (k,v) in states.items() if k in ('CA', 'FL') }
print(someStates)
```

19. Python Advanced Features - Lambda

Lambda Functions

A lambda is a small function without a name -- an anonymous function.

- It must return a result.
    - It can have only 1 expression without a return keyword.
    - The result will be automatically returned.

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

# lambda function syntax
# lambda input parameters | expression
adder = lambda a, b : a + b

print(type(add), add(3,4))
print(type(add), adder(3, 4))
```

Filtering with a lambda function

Python's filter() function is used to create a subset of a sequenced data (list or dictionaries).

- filter(func, data)
    - The first parameter is a function that returns a Boolean value. (any item that matches the condition will be returned as part of a new list)
    - The second parameter is a list or a dictionary.
    - The return value is a filter object. You need to convert the result back to a list or a dictionary.
    - The *filter()* a global function. You do not need to import any module.

```
def isOddNumber(number):
  return number % 2 == 1

numbers = [1,2,3,4,5,6,7,8,9,10]
oddNumbers = list(filter(isOddNumber, numbers))
print(oddNumbers)
```

With a lambda function, you do not need to create a separate function. And the code becomes clearer.

```
numbers = [1,2,3,4,5,6,7,8,9,10]
oddNumbers = list(filter(lambda n : n%2 == 1, numbers))
print(oddNumbers)
```

Filtering can be done using the list comprehension.

```
numbers = [1,2,3,4,5,6,7,8,9,10]
oddNumbers = [n for n in numbers if n%2 == 1]
print(oddNumbers)
```

The filter() function can be used with a dictionary.

```
fruits = {'apple':2, 'pear':1.5, 'mango':2.5, 'banana':0.9}

# use the dict() to conver the result
# in the lambda function f[0] = key, f[1] = value of each item
expensiveFruits = dict(filter(lambda f : f[1] >= 2, fruits.items()))
print(expensiveFruits)
```

You can get the same result using the dictionary comprehension.

```
fruits = {'apple':2, 'pear':1.5, 'mango':2.5, 'banana':0.9}
expensiveFruits = {k:v for (k,v) in fruits.items() if v >= 2}
print(expensiveFruits)
```

[Note]

- Use the comprehension – it is preferable.
- But you might see the code with the filter() function and you need to understand how it works.

Mapping with a lambda function

Python's map() function is used to transform a list to another list.

- map(func, data)
  - The first parameter is a function that returns an item for a new list or a dictionary.
  - The second parameter is a list or a dictionary.
  - The return value is a map object. You need to convert the result back to a list.
  - The map() is a global function. You do not need to import any module.

```
fruits = ['apple', 'pear', 'mango', 'banana']

lengths = map(lambda f : len(f), fruits)
print(lengths, type(lengths))

lengthsList = list(map(lambda f : len(f), fruits))
print(lengthsList, type(lengthsList))
```

[Note]

- Use the comprehension – it is preferable.
- But you might see the code with the map() function and you need to understand how it works.

- The only case you want to use the map() function is to combine 2 or more lists into a single list.

```
numbers1 = list(range(11))
numbers2 = list(range(10,0,-1))
print(numbers1, numbers2)

sumNumbers = list(map(lambda n1, n2: n1 + n2, numbers1, numbers2))
print(sumNumbers)
```

Reducing a sequence to a single item (value) with a lambda function

Python's reduce() function is used to accept a sequence of data (a list) and returns a single item.

- The first 2 items are passed to a provided function. The result is returned.
- The specified function is called with the previous result and the next item.
- The reduce() function is located in the functools module. You need to import the module first.

- Syntax: reduce(function, sequence, initialValue)
  - function: accepts 2 values and returns a single value
  - sequence: a data as a list
  - initialValue: optional. If this value is provided, the function is called with this initial value and the first item.

```
from functools import reduce

numbers = list(range(101))

# get the sum of 1-100 using the loop
total = 0
for n in numbers:
  total += n
print(total)

# using the reduce
total = reduce(lambda a,b: a+b, numbers)
print(total)
```

Here is another example – get the maximum value in a list.

```
from functools import reduce

numbers = [1, 33, 22, 12, 56, 3, 26, 45]

# get the maximum value
max = reduce(lambda a,b: a if a > b else b, numbers)
print(max)
```

20. Error Handling

Every programming language has a mechanism to handle errors or exceptions.

- An exception is a term to describe an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.

Examples of Exceptions/Errors

- A variable needs to be created before using it.

```
print(x) # NameError: name 'x' is not defined
```

- or you cannot divide a number by 0

```
y = 10 / 0  # ZeroDivisionError: division by zero
```

Handling the Error

- Put the code inside the try block
- When the code in the try block raises an error, the except block will be executed rather than crashing the application
- Without an except block, the program will crash when an error happens.

```
try:
  y = 10 / 0
except:
  print('Something bad happened.')
```

Many Exceptions

- You can define multiple except blocks for a special kind of error.
- Only the first matching except block will be executed.
    - Important! **Only 1 except block is executed**. (first matching)
- The *except* block without a special type name will be executed for any type of error.

```
try:
  y = 10 / 0
except ZeroDivisionError:
  print('You cannot divide a number by zero.')
except:
  print('Something bad happened.') # Not printed
```

```
try:
  print(x)  # Name Error
except ZeroDivisionError:
  print('You cannot divide a number by zero.')
except:
  print('Something bad happened.') # This block is executed.
```

Else and Finally

- When there is no error, the else block will be executed.
- The finally block will be executed regardless there is an error or not.
    - The finally block is executed all the time.

```
try:
  x = 10
  print(x)  # No error
except:
  print('Something bad happened.')
else:
  print('No Error.') # This block is executed.
finally:
  print('The end of the code.') # This block is executed.
```

```
try:
  print(x)  # error
except NameError:
  print('The undefined variable is used.') # This block is executed.
else:
  print('No Error.')
finally:
  print('The end of the code.') # This block is executed.
```

Accessing the Error/Exception objects

You can get an error object using the as keyword (followed by a variable name).

```
try:
  print (x)
except NameError as ne:
  print(ne)
```

```
try:
  y = 10 / 0
except ZeroDivisionError as zde:
  print(zde)
```

Raise an Exception - Advanced

You can raise a custom error using the raise keyword and Exception class.

```
number = int(input('Enter the number from 1 to 10: '))
try:
  if (number > 10):
    raise Exception('Your number is greater than 10.')
  elif (number < 1):
    raise Exception('Your number is less than 1.')
  print(number)
except Exception as e:
  print(e)
```

21. Handling Text Files

In Python, you can create, read, and update the files.

Files can be 2 types:

- Text files: have only text data – you can open the file in any text editor and read the content of it.
- Binary files: no text data, can be opened only in a specific applications that support the file types, images, or videos

Creating, Opening, and Closing a file

Python has an open() functions that creates and/or opens a file for reading or writing. The function is global. You do not need to import a module.

- *open(filename, mode)*
- mode:
  - r – Read: Default value. Opens a file for reading. Raises an error if the file does not exist
  - a – Append: Opens a file for appending and creates the file if it does not exist
  - w – Write: Opens a file for writing and creates the file if it does not exist
  - x – Create: Creates the specified file and returns an error if the file exists

After accessing the file, you must close the file.

- *file.close()*

```
# Create a file for writing
f = open("myfile.txt", "w")
print(type(f))

# Close a file -- when you finish with the file
f.close()
```

Writing Texts to a File

It is easy to write a text to file.

- *file.write()*

```
f = open("myfile.txt", "w")

# When the file is opened with "w" mode,
# all content is overwritten.
# The existing content will be removed.
f.write('Hello World!111')

f.close()
```

- The file must be closed. You can use the "with" sentence.
- If you open a file using a "with" block, the close() function is called automatically at the end of the block.

```
with open("myfile.txt", "w") as f:
  f.write('Good Morning!')
  # f.close() is called automatically
```

Reading Texts from a File

- *file.read()*
- *file.readline()*
- *file.readlines()*

Let's create a file and write a couple of lines of text before reading the file.

```
with open("myfile.txt", "w") as f:
  f.write('Hello\n')
  f.write('World\n')
  f.write('Good\n')
  f.write('Morning')
```

Reading everything in a file

```
with open("myfile.txt", "r") as f:
  content = f.read()
  print(content)
```

Reading text line by line

```
with open("myfile.txt", "r") as f:
  line1 = f.readline() # read a line including \n
  line2 = f.readline()
  print(line1)
  print(line2)
```

The readline() function reads a line of text but it includes the new line '\n' character.

```
with open("myfile.txt", "r") as f:
  line1 = f.readline() # read a line including \n
  line2 = f.readline()
  print(line1.strip()) # strip() also remove \n as well as a space
  print(line2.strip())
```

What if you do not know how many lines to read?

- Read everything line by line as a list

```
with open("myfile.txt", "r") as f:
  lines = f.readlines()  # it includes '\n'
  print(type(lines))
  print(lines)

  trimmedLines = [line.strip() for line in lines]
  print(trimmedLines)
```

There is a short-cut syntax without using the readlines() function.

```
with open("myfile.txt", "r") as f:
  trimmedLines = [line.strip() for line in f]
  print(trimmedLines)
```

Deleting a File

- Import the "os" module
- Use the *remove(filename)* function

```
import os

os.remove('myfile.txt')
```

Error Handling

What if the file does not exist when you try to open a file for reading?

```
with open("myfile.txt", "r") as f:
  content = f.read()  # FileNotFoundError
```

Using the try – except blocks

```
try:
  with open("myfile.txt", "r") as f:
    content = f.read()
except FileNotFoundError:
  print('The file does not exist.')
```

Example 1

Read fruit names from a file and then put them in a list.

```
with open("myfile.txt", "w") as f:
  f.write('Apple,Pear,Mango,Banana')

with open("myfile.txt", "r") as f:
  fruitsText = f.read()
  fruits = fruitsText.split(',')
  print(len(fruits))
  for fruit in fruits:
    print(fruit)
```

Example 2

Try to understand how the code works.

```
# module
class GreetMachine:
  def __init__(self, name):
    self._name = name

  def sayHello(self):
    return f'Hello, {self._name}'
```

```
# main
from mymodule import GreetMachine

with open("myfile.txt", "w") as f:
  f.write('Ddung\n')
  f.write('Hyang\n')
  f.write('Babo')

with open("myfile.txt", "r") as f:
  for line in f.readlines():
    greeting = GreetMachine(line.strip())
    print(greeting.sayHello())
```

22. OOP Advanced – Decorators and Properties

Python has a strange but interesting feature called decorators.

- Decorators change the behavior of your code by attaching @---
- Decorators are called meta-programming because they program the program!

Property Decorators

This section is the extension to the OOP –Encapsulation.

- Properties are the data part of an object.
- You can start the name of a property variable with an underscore to specify that the property can only be accessed inside the class code.
- To provide access to a property, you need to create functions – getters and setters.

```
# module
class Food():
  def __init__(self, name):
    self._name = name  # _name -- private property

  def getName(self):
    return self._name

  def setName(self, name):
    # the new name should have at least 3 characters
    if len(name) >= 3:
      self._name = name
```

```
# main
from mymodule import Food

p = Food('Bread')
print(p.getName()) # Bread

p.setName('Milk')
print(p.getName()) # Milk

p.setName('A')
print(p.getName()) # Still Milk
```

- Important! The getters and setters are methods. You need to use "()" to call methods.

Python provides a way to convert a getter or setter to act as a variable.

- For a getter, attach @property to the method
- For a setter, attach @---.setter to the method

```
# module
class Food():
  def __init__(self, name):
    self._name = name  # _name -- private property

  @property
  def name(self):
    return self._name

  @name.setter
  def name(self, name):
    # the new name should have at least 3 characters
    if len(name) >= 3:
      self._name = name
```

```
# main
from mymodule import Food

p = Food('Bread')
print(p.name) # Bread

p.name = 'Milk'
print(p.name) # Milk

p.name = 'A'
print(p.name) # Still Milk
```

- Important! The name is used like a variable.
- You do not need to use the property decorators. But it is good to know what they are because they are used in many built-in modules.

@staticmethod Decorators

Another strange conversion is to a method in a class.

- You need to create an object in order to call the method defined in a class code.
- Using a @staticmethod decorator, you can call the method without an object.
    - You can call the method directly by *className.methodName()*
    - A static method does not need the first parameter self.
- A static method is used when the information is the same for all objects.

```
# module
class BankAccount():
  def __init__(self, amount):
    self._amount = amount

  @property
  def amount(self):
    return self._amount

  def getTotal(self):
    return self._amount * (1 + BankAccount.getInterestRate())

  @staticmethod
  def getInterestRate():
   return 0.02 # 2 % for any accoount
```

Static method can be called without an object.

```
# main
from mymodule import BankAccount

print(BankAccount.getInterestRate()) # no need to crate an object
```

Check how the BankAccount class can be used.

```
# main
from mymodule import BankAccount

print(BankAccount.getInterestRate()) # no need to crate an object
account1 = BankAccount(1000)
account2 = BankAccount(2000)
print(account1.amount, account2.amount) # access the property
print(account1.getTotal(), account2.getTotal())
```

@classmethod Decorators

@classmethod is another decorator that can be called without an object.

- The first parameter is special, and it is not "*self*".
  - The first parameter represents a class, and you can use it to create an object.
- @classmethod is used to provide different ways to create an object using different parameters.

```
# module
class Person():
  def __init__(self, firstName, lastName):
    self._firstName = firstName
    self._lastName = lastName

  @property
  def fullName(self):
    return f'{self._firstName} {self._lastName}'

  @classmethod
  def createPersonFromFullName(cls, fullName):
    firstName, lastName = fullName.split(' ')
    return cls(firstName, lastName)

  @classmethod
  def createPersonFromList(cls, fullNameList):
    firstName, lastName = fullNameList
    return cls(firstName, lastName)
```

```
# main
from mymodule import Person

p1 = Person('A', 'B')
p2 = Person.createPersonFromFullName('C D')
p3 = Person.createPersonFromList(['E', 'F'])

print(p1.fullName)
print(p2.fullName)
print(p3.fullName)
```

23. datetime module

Using date and time in programming is tricky because it has many parts.

- The datetime class is used to represent the date and time in Python.
- The datetime class is located in the datetime module.

You need to import the datetime module.

```
import datetime as dt

print(dt.MINYEAR, dt.MAXYEAR) # 1 ~ 9999
print()
```

Use the now() method to get the current date and time.

- The now() method is a class method to create a datetime object using the current system time.

```
# the code is from the class definition

@classmethod
  def now(cls, tz=None):
    "Construct a datetime from time.time() and optional time zone info."
    t = _time.time()
    return cls.fromtimestamp(t, tz)
```

Access date and time data using the datetime properties.

```
# the code is from the class definition

# Read-only field accessors
@property
def year(self):
  """year (1-9999)"""
  return self._year
```

```
@property
  def month(self):
    """month (1-12)"""
    return self._month

@property
  def day(self):
    """day (1-31)"""
    return self._day
```

Here is the example of how to get the current date and time using the datetime object.

```
import datetime as dt

current = dt.datetime.now()
print(current, type(current))
print(current.year, current.month, current.day)
print(current.weekday())  # Monday == 0 ... Sunday == 6
print(current.hour, current.minute, current.second)
```

Create a datetime object with specific date and time.

```
import datetime as dt

b = dt.datetime(2000,12, 25)  # year, month, day
print(b)

c = dt.datetime(2000,12, 25, 11, 10, 59) # year, month, day, hour, minute, second
print(c)
```

In programming, it is important to know how to compute time difference (delta) between two datetime objects.

- The timedelta object is used.

```
# main
import datetime as dt

now = dt.datetime.now()
currentYear = now.year
past = dt.datetime(currentYear,1, 1) # Jan 1st of the current year

# timedelta object
delta = now - past
print(delta, type(delta))
print(delta.days)
```

Another example of timedelta object is the stopwatch.

```
import datetime as dt
import time

time1 = dt.datetime.now()

# wait for a while (3 seconds here)
time.sleep(3)

time2 = dt.datetime.now()

# timedelta object
delta = time2 - time1
print(delta.seconds)
print(delta.microseconds)
```

Final point is how to format the datetime object to a string.

- This is complex but important.
- The format uses the % code.
- `strftime()` method is used to specify the format.

- Year: %Y (full – 2020), %y (short – 20)
- Month: %B (full – December) %b (short – Dec) % m (number – 12)
- Day: %d (number – 31)

- Weekday: %A (full – Wednesday) %a (short – Wed)
- Hour: %H
- Minute: %M
- Second: %S

```
import datetime as dt

now = dt.datetime.now()

print(now)
print(now.strftime('%Y %B %d'))
print(now.strftime('%y-%b-%d %A'))
print(now.strftime('%B/%d/%Y %H:%M:%S'))
```

The full list is here. [datetime — Basic date and time types — Python 3.9.5 documentation](#).

24. Operator Overloading - Advanced

You can provide your own logic to check the object equality by modifying the __eq__() method.

- And then, you can compare 2 objects of the same class using the equality operator ==.

Python provides the mechanism to provide the mechanism to use any operator with your custom objects.

Let's start with the custom class.

- Get a value
- truncate the value into an integer
- stores the absolute value inside an object

```
# module
import math

class AbsoluteInteger:
  def __init__(self, value):
    self._value = abs(math.trunc(value)) # abs() is a global function

  @property
  def value(self):
    return self._value
```

```
# main
from mymodule import AbsoluteInteger

# all 5
n1 = AbsoluteInteger(5.2)
n2 = AbsoluteInteger(5.7)
n3 = AbsoluteInteger(-5.2)
n4 = AbsoluteInteger(-5.7)

print(n1.value, n2.value, n3.value, n4.value)
```

You can directly work on the value inside an object. But what if you are doing some operations directly with objects themselves.

```
# main
from mymodule import AbsoluteInteger

n1 = AbsoluteInteger(5)
n2 = AbsoluteInteger(5)

# uncomment the line by line and check the result
print(n1 == n2)  # False
print(n1 != n2)  # True

#print(n1 + n2)  # unsupported operand type +
#print(n1 - n2)  # unsupported operand type -
#print(n1 > n2)  # unsupported operand type >
#print(n1 < n2)  # unsupported operand type <
```

How do Operations on Objects Work

Programming is what you tell a computer to do. When you do some operations (==, +, -, > , or <), Python runtime calls the internal method to do the job.

- The internal method start and end with double underscores (__).

Here is an example: __eq__()

```
def __eq__(self, other):
        return self._value == other._value
```

```
print(n1 == n2)  # True
print(n1.__eq__(n2)) # True -- the same
```

Here is how the operation works:

- n1 == n2
  - Start from the object in the left: n1
  - Call the __eq__ method on n1
    - Pass the object in the right: n2 – as an argument of the method
  - The method returns a Boolean value (True or False)

Implement Operator Magic Methods

In Python, you can override the default behavior of the operators by implementing so called magic methods.

- Comparison Operators

Operator	Magic Method
==	__eg__(self, other)
!=	__ne__(self, other)
<	__lt__(self, other)
>	__gt__(self, other)
<=	__le__(self, other)
>=	__ge__(self, other)

```
# module
import math

class AbsoluteInteger:
  def __init__(self, value):
    self._value = abs(math.trunc(value)) # abs() is a global function

  @property
  def value(self):
    return self._value

  def __eq__(self, other):
    return self._value == other._value
```

```
  def __ne__(self, other):
    return self._value != other._value

  def __lt__(self, other):
    return self._value < other._value

  def __gt__(self, other):
    return self._value > other._value

  def __le__(self, other):
    return self._value <= other._value

  def __ge__(self, other):
    return self._value >= other._value
```

```
# main
from mymodule import AbsoluteInteger

n1 = AbsoluteInteger(5) # 5
n2 = AbsoluteInteger(-6) # 6

print(n1 == n2)  # 5 == 6 False
print(n1 != n2)  # 5 != 6 True
print(n1 < n2)  # 5 < 6 True
print(n1 > n2)  # 5 > 6 False
print(n1 <= n2)  # 5 <= 6 True
print(n1 >= n2)  # 5 >= 6 False
```

- Arithmetic Operators

Operator	Magic Method
+	__add__(self, other)
-	__sub__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)
//	__floordiv__(self, other)
%	__mod__(self, other)
**	__pow__(self, other)

Implementing arithmetic operators needs to be more careful.

- Unlike comparison operators, which return a Boolean value, the arithmetic operators can return anything: numbers, strings, or any kind of object.
- **In general, return the same type of object.**

```
# module
import math

class AbsoluteInteger:
  def __init__(self, value):
    self._value = abs(math.trunc(value)) # abs() is a global function

  @property
  def value(self):
    return self._value

  def __add__(self, other):
    return AbsoluteInteger(self._value + other._value)

  def __sub__(self, other):
    return AbsoluteInteger(self._value - other._value)

  def __mul__(self, other):
    return AbsoluteInteger(self._value * other._value)

  def __truediv__(self, other):
    return AbsoluteInteger(self._value / other._value)

  def __floordiv__(self, other):
    return AbsoluteInteger(self._value // other._value)

  def __mod__(self, other):
    return AbsoluteInteger(self._value % other._value)

  def __pow__(self, other):
    return AbsoluteInteger(self._value ** other._value)
```

```
# main
from mymodule import AbsoluteInteger

n1 = AbsoluteInteger(-5) # 5
n2 = AbsoluteInteger(2) # 2
```

```
print(n1 + n2)  # object
print((n1 + n2).value)  # 7
print((n1 - n2).value)  # 3
print((n1 * n2).value)  # 10
print((n1 / n2).value)  # 2  careful (2.5 -> 2)
print((n1 // n2).value)  # 2
print((n1 % n2).value)  # 1
print((n1 ** n2).value)  # 25
```

25. Working with Database with SQLite

Python provides the lightweight database engine SQLite.

- The "sqlite3" module is provided by default.

Creating and accessing the database

- It is easy to create a new database or connect to an existing database using the connect() method.
- You need to get a cursor from the *connection* object to interact with a database.

```
import sqlite3

# create a connection
connection = sqlite3.connect('employee.db')
print(type(connection) )

# get a cursor
cursor = connection.cursor()
print(type(cursor) )

# working with db using the cursor

# close the cursor
cursor.close()

# close the connection
connection.close()
```

Creating a Table and inserting data

- Database consists of tables. (like an Excel worksheet)

- The first step is to create a table by providing a structure. (what kind of data can be stored)
    - A table consists of columns.

    - You need to provide the data type for each column.
    - SQLite provides the following data types:
        - TEXT
        - INTEGER
        - REAL (floating point values)
    - The "CREATE" statement is used to create a table
- And then, you can add some data to the table.
    - The "INSERT" statement is used to insert data into a table.
- The statements are executed using the execute() method of a cursor object.
- To save the data into a table, you need to confirm your actions by calling the connection.commit().

```
import sqlite3

connection = sqlite3.connect('order.db')
cursor = connection.cursor()

# working with db using the cursor
cursor.execute('''
CREATE TABLE IF NOT EXISTS orders (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  price REAL NOT NULL,
  quantity INTEGER NOT NULL,
  date TEXT NOT NULL
)
''')

# We can add a set of data matched with a table structure
# Each set of data is called a row.
cursor.execute("INSERT INTO orders (id, name, price, quantity, date) VALUES (1, '
Bread 1Lb', 2.99, 2, '2020-12-31')")
cursor.execute("INSERT INTO orders (id, name, price, quantity, date) VALUES (2, '
Milk 4L', 5.99, 1, '2021-1-1')")
cursor.execute("INSERT INTO orders (id, name, price, quantity, date) VALUES (3, '
Eggs 12', 3.99, 1, '2020-2-27')")

# Need to confirm your actions to save data into a database
connection.commit()

cursor.close()
connection.close()
```

Read data from a database

- You need to use the "SELECT" statement to read data from tables

One way to access the table is to read all data (rows) at one time.

```
import sqlite3

connection = sqlite3.connect('order.db')
cursor = connection.cursor()

# working with db using the cursor
cursor.execute('SELECT * FROM orders')
allData = cursor.fetchall()
print(allData, type(allData)) # list of tuples

for index1 in range(len(allData)):
  row = allData[index1]
  print('\t', row, type(row))
  for index2 in range(len(row)):
    value = row[index2]
    print('\t\t', value, type(value))

cursor.close()
connection.close()
```

Another way is to read each row one by one.

```
import sqlite3

connection = sqlite3.connect('order.db')
cursor = connection.cursor()
# working with db using the cursor
for row in cursor.execute('SELECT * FROM orders'):
  print(row, type(row))
  for index2 in range(len(row)):
    value = row[index2]
    print('\t', value, type(value))

cursor.close()
connection.close()
```

Read data using the row factory

Retrieving a value in a row using an index can be tedious and not clear which value we are accessing.

- Set the connection's row_factory option to be sqlite3.Row.
- Now you can access the value using the column name.

```
import sqlite3

connection = sqlite3.connect('order.db')
connection.row_factory = sqlite3.Row
cursor = connection.cursor()

# working with db using the cursor
for row in cursor.execute('SELECT * FROM orders'):
  print(row, type(row))
  print('\t', row['id'])
  print('\t', row['name'])
  print('\t', row['price'])
  print('\t', row['quantity'])
  print('\t', row['date'])

cursor.close()
connection.close()
```