01076105, 01075106

Object Oriented Programming

Object Oriented Programming Project

Polymorphism

Polymorphism

- มาจากคำว่า “poly” แปลว่าหลาย กับ “morph” แปลว่าเปลี่ยน รวมแล้วคือ เปลี่ยนได้หลายแบบ
- ใน OOP คำนี้หมายถึง การที่ 1 operation หรือ method สามารถใช้งานกับ Object ที่หลากหลายได้ เช่น จากรูป จะเห็นได้ว่า operator * สามารถใช้ได้กับ ตัวเลข string list โดยมีพฤติกรรมที่แตกต่างกันไปในแต่ละประเภท
- สิ่งที่สำคัญ คือ **Interface** ต้องเหมือนกัน

```
l = [2, "1", 3, 4, [1]]

for shape in l:
    print(shape*2)
```

Polymorphism

- ตัวอย่าง คือ การเล่นไฟล์เพลง audio_file.play() (เพิ่มประเภทอื่นๆ ได้)

```python
class AudioFile:
    def __init__(self, filename):
        if not filename.endswith(self.ext):
            raise Exception("Invalid file format")
        self.filename = filename

class MP3File(AudioFile):
    ext = "mp3"
    def play(self):
        print("playing {} as mp3".format(self.filename))

class WavFile(AudioFile):
    ext = "wav"
    def play(self):
        print("playing {} as wav".format(self.filename))

mp3 = MP3File("myfile.mp3")
wav = WavFile("myfile.wav")
mp3.play()
wav.play()
```

Polymorphism

- จะเห็นว่าไฟล์ทุกประเภท จะมีการตรวจสอบชื่อไฟล์ ที่ Base Class ทั้งหมด ซึ่งแสดงว่า Base Class สามารถจะเข้าถึง ext ใน Subclass ได้
- แต่ใน method play() จะแยกกันทำงานในไฟล์แต่ละประเภท
- ดังนั้นไฟล์แต่ละประเภท จึงถูกจัดการด้วยวิธีการที่แตกต่างกัน นี่เป็นความสามารถของ Polymorphism
- ความสามารถ Polymorphism จะเกิดขึ้นได้ ต้องมี Inheritance มาก่อน
- เราอาจกล่าวได้ว่า Polymorphism คือการใช้ Interface ร่วมกันของข้อมูลที่แตกต่างกัน

Polymorphism : magic method

- เราอาจออกแบบให้ Class ของเรา ตอบสนองกับ เครื่องหมาย +-*/ หรือ in
- หรือตอบสนองกับ subscript หรือ slice หรือ loop (คล้ายกับ list) ได้
- ฟังก์ชัน __add__ อยู่ในกลุ่มที่เรียกว่า magic method หรือ dunder (double under)

```
>>> var1 = 'hello '
>>> var2 = 'world'
>>> var1 + var2
'hello world'
>>> var1.__add__(var2)
'hello world'
```

Polymorphism : magic method

```python
class Length:
    __metric = {"mm" : 0.001, "cm" : 0.01, "m" : 1, "km" : 1000,
                "in" : 0.0254, "ft" : 0.3048, "yd" : 0.9144,
                "mi" : 1609.344 }
    def __init__(self, value, unit = "m" ):
        self.value = value
        self.unit = unit
    def Converse2Metres(self):
        return self.value * Length.__metric[self.unit]
    def __add__(self, other):
        l = self.Converse2Metres() + other.Converse2Metres()
        return Length(l / Length.__metric[self.unit], self.unit )
    def __str__(self):
        return str(self.Converse2Metres())
    def __repr__(self):
        return "Length(" + str(self.value) + ", '" + self.unit + "')"
```

Polymorphism : magic method

- ใน __init__ จะรับมูลความยาวและหน่วย หากไม่กำหนดจะให้เป็น m
- ฟังก์ชัน Converse2Metres ทำหน้าที่แปลงเป็นเมตร
- ฟังก์ชัน __add__ ทำหน้าที่บวกระหว่าง Object แบบเดียวกันและส่งกลับเป็น Object
- ฟังก์ชัน __str__ ทำหน้าที่แสดงข้อมูลที่เก็บที่แปลงเป็นเมตรแล้ว
- ฟังก์ชัน __repr__ จะแสดงความยาวในหน่วยที่กำหนด

Polymorphism : magic method

- ผลการทำงาน

```
x = Length(4)
print(x)
z = Length(4.5, "yd") + Length(1)
print(repr(z))
print(z)
```

```
4
Length(5.593613298337708, 'yd')
5.1148
```

Polymorphism : magic method

- จากโปรแกรม เนื่องจาก method __add__ จะรับข้อมูลเป็น object จึงไม่สามารถไปบวกกับข้อมูล type อื่นได้ เช่น เขียนว่า

```
x  = Length(1) + 1
```

- ก็จะเกิด Error ดังนั้นจะแก้ไข __add__ ใหม่ ก็จะทำงานได้

```
    def __add__(self, other):
        if type(other) == int or type(other) == float:
            l = self.Converse2Metres() + other
        else:
            l = self.Converse2Metres() + other.Converse2Metres()
        return Length(l / Length.__metric[self.unit], self.unit)
```

Polymorphism : magic method

- แต่หากเขียนตามนี้ ก็ไม่สามารถทำงานได้ เนื่องจาก ใช้กับ += ไม่ได้

```
x  += Length(1)
```

- กรณีนี้ให้ใช้ __radd__

```
def __radd__(self, other):
    if type(other) == int or type(other) == float:
        l = self.Converse2Metres() + other
    else:
        l = self.Converse2Metres() + other.Converse2Metres()
    return Length(l / Length.__metric[self.unit], self.unit)
```

- แม้ในโปรแกรมจะเขียนเหมือนกัน แต่ผลการทำงานจะต่างกัน

Polymorphism : magic method

Binary Operators

Operator	Method
+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
//	object.__floordiv__(self, other)
/	object.__truediv__(self, other)
%	object.__mod__(self, other)
**	object.__pow__(self, other[, modulo])
<<	object.__lshift__(self, other)
>>	object.__rshift__(self, other)
&	object.__and__(self, other)
^	object.__xor__(self, other)
|	object.__or__(self, other)

Extended Assignments

Operator	Method
+=	object.__iadd__(self, other)
-=	object.__isub__(self, other)
*=	object.__imul__(self, other)
/=	object.__idiv__(self, other)
//=	object.__ifloordiv__(self, other)
%=	object.__imod__(self, other)
**=	object.__ipow__(self, other[, modulo])
<<=	object.__ilshift__(self, other)
>>=	object.__irshift__(self, other)
&=	object.__iand__(self, other)
^=	object.__ixor__(self, other)
|=	object.__ior__(self, other)

Polymorphism : magic method

Unary Operators

Operator	Method
-	object.__neg__(self)
+	object.__pos__(self)
abs()	object.__abs__(self)
~	object.__invert__(self)
complex()	object.__complex__(self)
int()	object.__int__(self)
long()	object.__long__(self)
float()	object.__float__(self)
oct()	object.__oct__(self)
hex()	object.__hex__(self

Comparison Operators

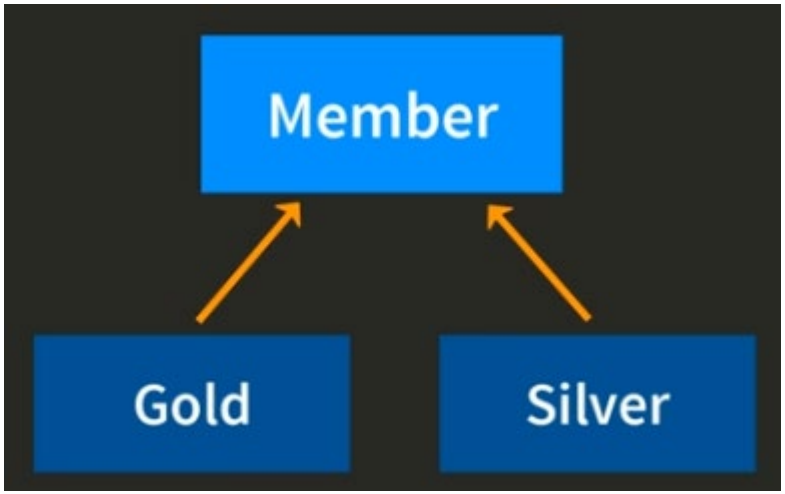
Operator	Method
<	object.__lt__(self, other)
<=	object.__le__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)
>=	object.__ge__(self, other)
>	object.__gt__(self, other)

Polymorphism : magic method

- __getitem__ ใช้สำหรับดึงข้อมูลที่กำหนด
- __getslice__ ใช้สำหรับดึงข้อมูลในช่วงที่กำหนด
- __contains__ ใช้สำหรับ operator “in”
- https://rszalski.github.io/magicmethods/
- https://www.analyticsvidhya.com/blog/2021/08/explore-the-magic-methods-in-python/

Abstract Base Class

- เป็น Class ที่ไม่ใช้สำหรับสร้าง Instance แต่เป็นโครงสำหรับ Class อื่นที่สืบทอดไป โดยคลาสที่สืบทอดไป ต้อง implement ตามที่กำหนด
- สมมติว่ามีระบบสมาชิก โดยแบ่งเป็น 2 ประเภท คือ Gold กับ Silver



Abstract Base Class

```python
from abc import ABC, abstractmethod

class Member(ABC):
    def __init__(self, m_id, fname, lname):
        self.m_id = m_id
        self.fname = fname
        self.lname = lname

    @abstractmethod
    def discount(self):
        pass

    def full_name(self):
        return "{} {}".format(self.fname, self.lname)

class Gold(Member):
    def discount(self):
        return .10

class Silver(Member):
    def discount(self):
        return .05
```

Abstract Base Class

- Class MediaLoader จะทำหน้าที่เป็น Base Class โดยบังคับให้ต้องกำหนด Attribute ext และ Method play

```python
import abc

class MediaLoader(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def play(self):
        pass

    @abc.abstractproperty
    def ext(self):
        pass
```

Abstract Base Class

- จะเห็นว่า Class Ogg ทำการ Inherit มาจาก MediaLoader
- ต้องมีการกำหนด ext และ def play มิฉะนั้นจะ Error

```
class Ogg(MediaLoader):
    ext = '.ogg'

    def play(self):
        pass

o = Ogg()
```

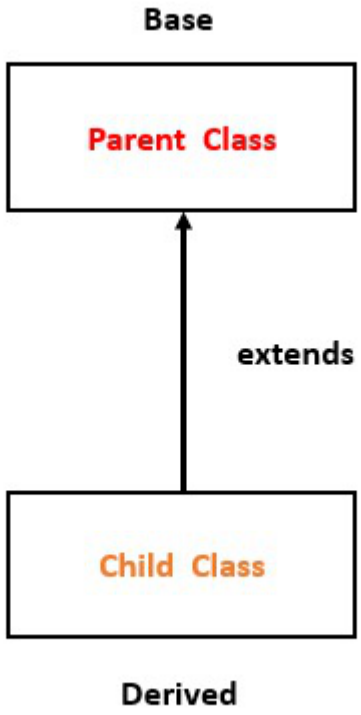
“with” Class

- เราสามารถใช้ with กับ Class ได้

```
class MyClass(object):
    def __enter__(self):
        print('We have entered "with"')
        return self

    def __exit__(self, type, value, traceback):
        print('error type: {0}'.format(type))
        print('error value: {0}'.format(value))
        print('error traceback: {0}'.format(traceback))
        print('We have exit "with"')

    def sayhi(self):
        print('hi, instance %s' % (id(self)))

with MyClass() as cc:
    cc.sayhi()

print('after the "with" block')
```

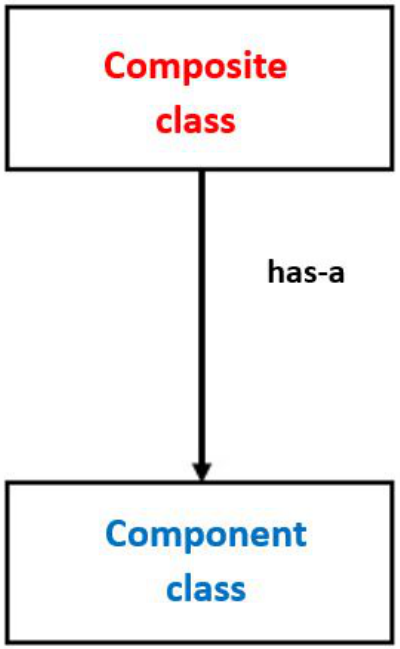
Composition

- ใน Inheritance จะเป็นความสัมพันธ์แบบ Is-A Relation คือ ใน Subclass เป็นชนิดย่อยของ Super Class



Composition

- ใน Composition จะเป็นความสัมพันธ์แบบ Has-A Relation
- โดยใน Class composite จะมี Class component เป็นส่วนประกอบ



Composition

```
class Component:
    def __init__(self):          # composite class constructor
        print('Component class object created...')

    def m1(self):                # composite class instance method
        print('Component class m1() method executed...')

class Composite:
    def __init__(self):          # composite class constructor

        self.obj1 = Component() # creating object of component class
        print('Composite class object also created...')

    def m2(self):                # composite class instance method
        print('Composite class m2() method executed...')
        self.obj1.m1()           # calling m1() method of component class

obj2 = Composite()
obj2.m2()    # calling m2() method of composite class
```

Composition

```python
class Salary:
    def __init__(self, monthly_income):
        self.monthly_income = monthly_income

    def get_total(self):
        return (self.monthly_income * 12)


class Employee:
    def __init__(self, monthly_income, bonus):
        self.monthly_income = monthly_income
        self.bonus = bonus
        self.obj_salary = Salary(self.monthly_income)

    def annual_salary(self):
        return "Total: " + str(self.obj_salary.get_total() + self.bonus) + ' bath'


obj_emp = Employee(2600, 500)
print(obj_emp.annual_salary())
```

Composition

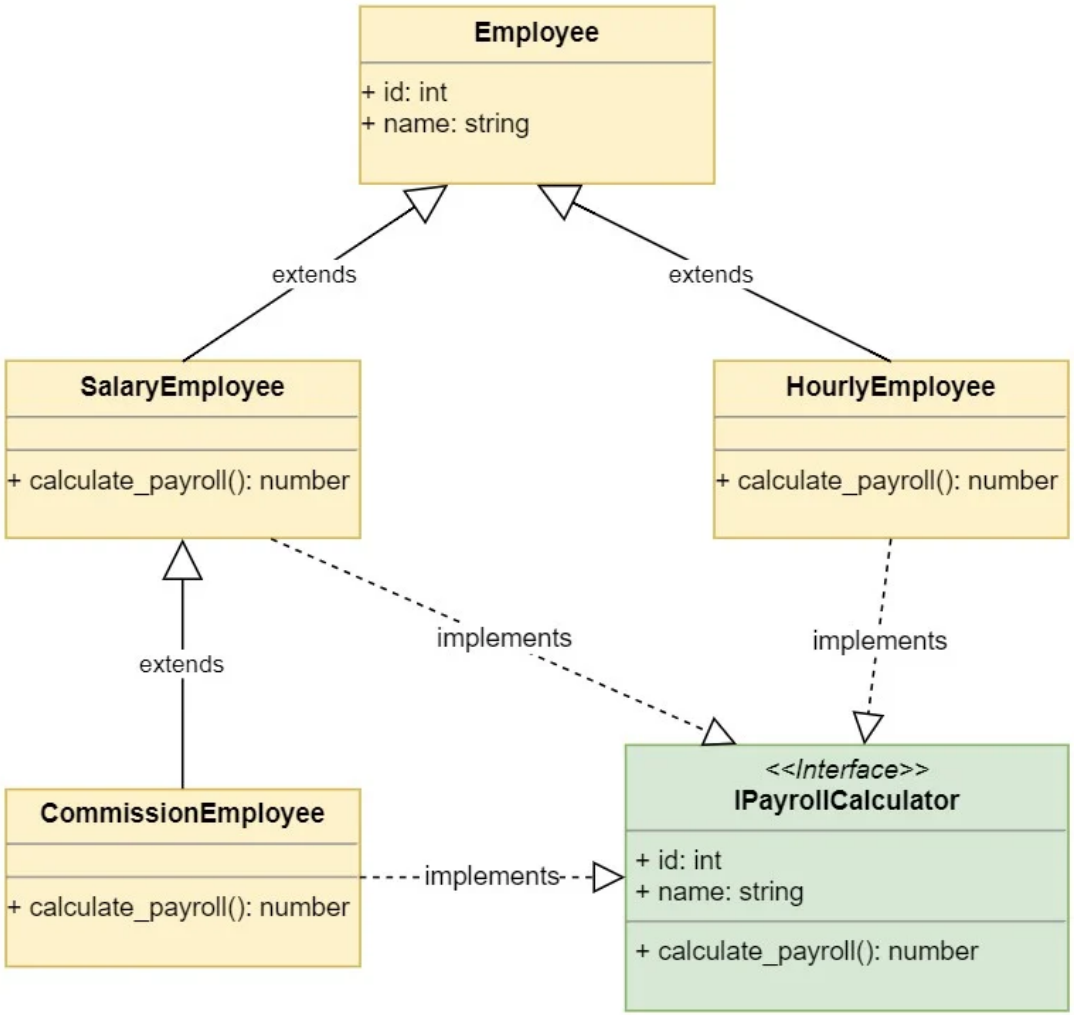
```python
class Animal:
    name = ""
    category = ""

    def __init__(self, name):
        self.name = name

    def set_category(self, category):
        self.category = category

class Falcon(Animal):
    category = "birds"

class Parrots(Animal):
    category = "birds"
```

Composition

```python
class Zoo:
    def __init__(self):
            self.current_animals = {}

    def add_animal(self, animal):
        self.current_animals[animal.name] = animal.category

    def total_of_category(self, category):
        result = 0
        for animal in self.current_animals.values():
            if animal == category:
                result += 1
        return result

zoo = Zoo()
falcon = Falcon("Falcon")  # create an instance of the Falcon class
parrots = Parrots("Parrots")  # create an instance of the Parrots class
zoo.add_animal(falcon)
zoo.add_animal(parrots)
print(zoo.total_of_category("birds"))  # how many zoo animal types in the birds category
```

Composition

- Composition อาจใช้ในกรณีที่หลายๆ Class ใช้ Interface แบบเดียวกัน จึงสร้าง Component Class เอาไว้เพื่อบริการหลายๆ Composition Class



Iterator, Iterable

- Iterable หมายถึง โครงสร้างข้อมูลที่มีลักษณะเป็นชุด และใช้กับการวน loop ได้ เช่น List, Tuple, Dictionary, String, File, Generator
- Object ใดๆ จะเป็น Iterable ใน Class จะต้องมี method __Iter__
- เราสามารถตรวจสอบว่า Class มีการ Implement method ใดบ้าง โดยใช้คำสั่ง dir

```
>>> L = [1, 2, 3]
>>> dir(L)
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__', '__dir
__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__
hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__',
 '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed_
_', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append
', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Iterator, Iterable

- เมื่อมีการ Loop เกิดขึ้น จะมีการเรียกใช้ __iter__ ใน Object และมีการส่งกลับเป็น Iterator เพื่อใช้ในการวน Loop ต่อไป
- ดังนั้น Iterator จะต้องสามารถเรียกข้อมูลตัวต่อไปได้

```
>>> L = [1, 2, 3]
>>> next(L)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list' object is not an iterator
>>>
```

จะเห็นว่า List ไม่ใช่ iterator

```
>>> L = [1, 2, 3]
>>> i_num = L.__iter__()
>>> dir(i_num)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__length_hint__', '__lt__', '__ne__', '__new__', '__next__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__', '__sizeof__', '__str__', '__subclasshook__']
>>> next(i_num)
1
>>>
```

Iterator, Iterable

- เราสามารถสร้าง Iterator ขึ้นมาใช้งานได้

```python
class MyRange:
    def __init__(self, start, end):
        self.value = start
        self.end = end

    def __iter__(self):
        return self

    def __next__(self):
        if self.value >= self.end:
            raise StopIteration
        current = self.value
        self.value += 1
        return current

nums = MyRange(1,10)
for num in nums:
    print(num)
```

Generator

- สมมติว่ามีโปรแกรมต่อไปนี้
- เป็นโปรแกรมที่สร้างลำดับที่เป็นเลขยกกำลัง 2

```python
def square_numbers(nums):
    result = []
    for i in nums:
        result.append(i*i)
    return result

my_nums = square_numbers([1,2,3,4,5])
print(my_nums)
```

Generator

- ถ้าเราแก้โปรแกรมเป็นดังนี้ (ความหมายของ yield คือ ผลิต)
- เมื่อเรา print(my_nums) จะบอกว่าเป็น generator

```
def square_numbers(nums):
    for i in nums:
        yield (i*i)

my_nums = square_numbers([1,2,3,4,5])
print(my_nums)
```

```
<generator object square_numbers at 0x00000235AF1AEA40>
```

Generator

- การใช้ Generator จะมีข้อดี คือ หากใช้แบบ return ข้อมูลมาทั้ง list จะทำให้เปลืองเนื้อที่หน่วยความจำ แต่หากใช้เป็น Generator จะส่งข้อมูลในลำดับมาทีละ 1 ค่า เท่านั้น
- การดึงข้อมูลมาใช้ จะใช้คำสั่ง next ตามตัวอย่าง
- ดังนั้นกรณีที่มีข้อมูลเป็นชุดจำนวนมากๆ การใช้ Generator จะดีกว่า เร็วกว่า

```
my_nums = square_numbers([1,2,3,4,5])
print(next(my_nums))
print(next(my_nums))
print(next(my_nums))
```

```
1
4
9
```

Itertools

- เป็นโมดูลที่มี Generator ที่มีการใช้งานบ่อยๆ
- Iterator อยู่ 2 ประเภทคือ Infinite Iterators และ Finite Iterators หรือก็คือ Iterator ที่วนได้ไม่มีวันสิ้นสุด กับ วนแบบมีจุดสิ้นสุด
- Itertools.count เป็นตัวนับ ทำหน้าที่สร้างตัวเลข โดยกำหนดตัวเริ่มต้นและ step ได้ การใช้งานต้องระวัง เพราะเป็นประเภท Infinit

```python
import itertools
for counter in itertools.count(start=0, step=1):
    print(counter)
```

Itertools

- Itertools.cycle เป็นตัวที่นำข้อมูลมาวนซ้ำ

```
cyc = itertools.cycle(range(2,7,2))
i = 0
while i<30:
    print(next(cyc),end=';')
    i += 1

# ได้ 2;4;6;2;4;6;2;4;6;2;4;6;2;4;6;2;4;6;2;4;6;2;4;6;2;4;6;
```

- Itertools อื่นๆ สามารถดูได้ที่ https://docs.python.org/3/library/itertools.html

Lambda Function

- เป็นฟังก์ชันที่ไม่มีชื่อ

```
c2f = lambda c: (c * (9 / 5)) + 32
print (c2f(100))
```

- ปกติ Lambda มักใช้กับ map เพื่อประมวลผลข้อมูลใน List หรืออื่นๆ

```
t=[0,10,20,30,100]
f=list(map(lambda c: c2f(c), t))
print(f)
```

```
num1 = [4, 5, 6]
num2 = [5, 6, 7]
result = map(lambda n1, n2: n1+n2, num1, num2)
print(list(result))
```

Lambda Function

- การส่งค่ากลับ (return) จากฟังก์ชันได้เพียงค่าเดียวเท่านั้น
- ไม่สามารถเรียกใช้เพื่อให้แสดงผล (print) ค่าข้อมูลได้
- ตัวแปรที่ใช้ในการประมวลผลของฟังก์ชันแบบไม่ระบุชื่อ จะอยู่ภายในขอบเขตพื้นที่ที่ใช้ สำหรับเก็บตัวแปร (namespace) ของตัวเองเท่านั้น และไม่สามารถเรียกข้าม namespace ได้

Map function

- ทำหน้าที่นำแต่ละข้อมูลที่อยู่ใน iterable (list, tuple etc.) ไปส่งเข้าฟังก์ชันที่ระบุ และส่งกลับเป็น iterator (ใช้ลดการใช้ loop)

```python
numbers = [2, 4, 6, 8, 10]

def square(number):
  return number * number

# apply square() function to each item of the numbers list
squared_numbers_iterator = map(square, numbers)

# converting to list
squared_numbers = list(squared_numbers_iterator)
print(squared_numbers)

# Output: [4, 16, 36, 64, 100]
```

Map function

- ตัวอย่างนี้ เป็นการนำ Iterator จำนวน 2 ตัว ส่งเข้าไปยัง lambda function

```
num1 = [4, 5, 6]
num2 = [5, 6, 7]

result = map(lambda n1, n2: n1+n2, num1, num2)

print(list(result))
```

```
[9, 11, 13]
```

Filter Function

- คล้ายกับฟังก์ชัน Map แต่ Filter() จะทำการ Return Elements ที่ Function ที่ใช้งาน Return เป็นค่า True เท่านั้น

```
letters = ['a', 'b', 'd', 'e', 'i', 'j', 'o']

# a function that returns True if letter is vowel
def filter_vowels(letter):
    vowels = ['a', 'e', 'i', 'o', 'u']
    return True if letter in vowels else False

filtered_vowels = filter(filter_vowels, letters)

# converting to tuple
vowels = tuple(filtered_vowels)
print(vowels)
```

Filter Function

- ใช้ filter ร่วมกับ lambda function จะทำให้โปรแกรมสั้นลง

```
numbers = [1, 2, 3, 4, 5, 6, 7]

# the lambda function returns True for even numbers
even_numbers_iterator = filter(lambda x: (x%2 == 0), numbers)

# converting to list
even_numbers = list(even_numbers_iterator)

print(even_numbers)
```

Filter Function

```
def interSection(arr1,arr2):

    # filter(lambda x: x in arr1, arr2) -->
    # filter element x from list arr2 where x
    # also lies in arr1
    result = list(filter(lambda x: x in arr1, arr2))
    print ("Intersection : ",result)

arr1 = [1, 3, 4, 5, 7]
arr2 = [2, 3, 5, 6]
interSection(arr1, arr2)
```

Exception

- โปรแกรมที่ดีควรจะให้การทำงานที่ถูกต้องเสมอ แต่บางครั้งก็อาจเกิดความผิดพลาดได้ เช่น จาก Input ที่ป้อนเข้ามาไม่ตรงตามกำหนด หรือ access ข้อมูลเกินจำนวนที่มี
- แนวทางแก้ปัญหามี 2 แนวทาง คือ 1) ตรวจสอบ Input ในทุกๆ การป้อน เพื่อให้แน่ใจว่าขอบเขตของ Input เป็นไปตามที่กำหนด และ 2) คือใช้ exception
- Exception คือ เหตุการณ์ที่เกิดขึ้น เมื่อโปรแกรมไม่ทำงานไปตามที่กำหนด
- ใน python นั้น Exception ก็เป็น Object เช่นกัน

Exception

- ใน python เมื่อมีปัญหากับการทำงาน จะสร้าง exception ขึ้น เช่น

```
print "hello word"
```

```
  File "C:\Users\khtha\AppData\Local\Temp/ipykernel_12568/345270128.py",
    print "hello word"
          ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print("hello word")?
```

- หรือหารด้วย 0

```
x = 5 / 0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_12568/1806623527.py in <module>
----> 1 x = 5 / 0

ZeroDivisionError: division by zero
```

Exception

- Index Error

```
lst = [1,2,3]
print(lst[3])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_12568/1052905902.py in <module>
      1 lst = [1,2,3]
----> 2 print(lst[3])

IndexError: list index out of range
```

Exception

- Type Error

```
lst = [1,2,3]
lst + 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_12568/3418216332.py in <module>
      1 lst = [1,2,3]
----> 2 lst + 2

TypeError: can only concatenate list (not "int") to list
```

Exception

- Attribute Error

```
lst = [1,2,3]
lst.add
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_12568/453125606.py in <module>
      1 lst = [1,2,3]
----> 2 lst.add

AttributeError: 'list' object has no attribute 'add'
```

Exception

- Key Error

```
d = {'a': 'hello'}
d['b']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_12568/3661029466.py in <module>
      1 d = {'a': 'hello'}
----> 2 d['b']

KeyError: 'b'
```

- Name Error

```
print(this_is_not_a_var)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_12568/2762618246.py in <module>
----> 1 print(this_is_not_a_var)

NameError: name 'this_is_not_a_var' is not defined
```

Exception

- สมมติว่าเราสร้าง Class ที่ extend จาก List เราสามารถตรวจสอบ Error และสามารถ raise exception ได้ ตามประเภทของ exception

```
class EvenOnly(list):
    def append(self, integer):
        if not isinstance(integer, int):
            raise TypeError("Only integers can be added")
        if integer % 2:
            raise ValueError("Only even numbers can be added")
        super().append(integer)
```

Exception

- เมื่อเกิด exception ขึ้น โปรแกรมจะหยุดทำงานในบรรทัดต่อๆ ไป
- ดูจากตัวอย่าง

```
def no_return():
    print("I am about to raise an exception")
    raise Exception("This is always raised")
    print("This line will never execute")
    return "I won't be returned"

no_return()
```

```
I am about to raise an exception

---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_12568/2628487590.py in <module>
      5     return "I won't be returned"
      6
----> 7 no_return()

~\AppData\Local\Temp/ipykernel_12568/2628487590.py in no_return()
      1 def no_return():
      2     print("I am about to raise an exception")
----> 3     raise Exception("This is always raised")
      4     print("This line will never execute")
      5     return "I won't be returned"

Exception: This is always raised
```

Exception Handling

- ในการจัดการกับ exception จะใช้ try.. catch จะเป็นการดัก error จาก python ให้มาอยู่ในส่วนที่โปรแกรมควบคุมได้ และ ทำงานต่อไป

```
def no_return():
    print("I am about to raise an exception")
    raise Exception("This is always raised")
    print("This line will never execute")
    return "I won't be returned"

try:
    no_return()
except:
    print("I caught an exception")
    print("executed after the exception")
```

Exception Handling

- การดัก error ต้องทำให้ครอบคลุมทุกกรณี เช่น

```
def funny_division(divider):
    try:
        return 100 / divider
    except ZeroDivisionError:
        return "Zero is not a good idea!"

print(funny_division(0))
print(funny_division(50.0))
print(funny_division("hello"))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_12568/200746621.py in <module>
      7 print(funny_division(0))
      8 print(funny_division(50.0))
----> 9 print(funny_division("hello"))
```

Exception Handling

- โปรแกรมนี้ยังมี error เกิดขึ้นหรือไม่

```
def funny_division2(anumber):
    try:
        if anumber == 13:
            raise ValueError("13 is an unlucky number")
        return 100 / anumber
    except (ZeroDivisionError, TypeError):
        return "Enter a number other than zero"

for val in (0, "hello", 50.0, 13):
    print("Testing {}:".format(val), end=" ")
    print(funny_division2(val))
```

Exception Handling

- Version 3

```
def funny_division3(anumber):
    try:
        if anumber == 13:
            raise ValueError("13 is an unlucky number")
            return 100 / anumber
    except ZeroDivisionError:
        return "Enter a number other than zero"
    except TypeError:
        return "Enter a numerical value"
    except ValueError:
        print("No, No, not 13!")
        raise
```

Exception Handling

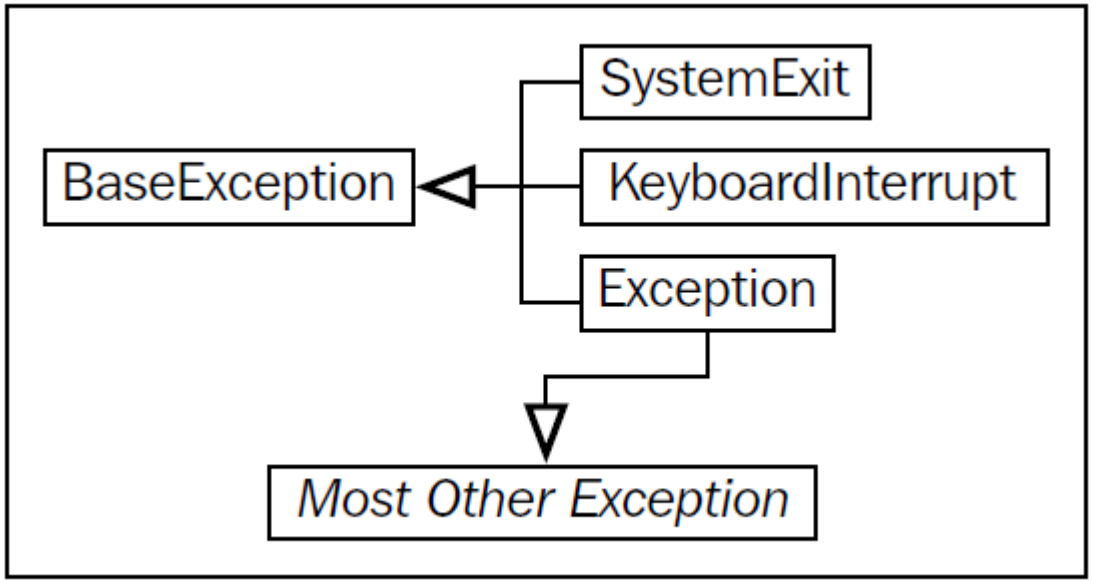
```python
import random
some_exceptions = [ValueError, TypeError, IndexError, None]

try:
    choice = random.choice(some_exceptions)
    print("raising {}".format(choice))
    if choice:
        raise choice("An error")
except ValueError:
    print("Caught a ValueError")
except TypeError:
    print("Caught a TypeError")
except Exception as e:
    print("Caught some other error: %s" % ( e.__class__.__name__))
else:
    print("This code called if there is no exception")
finally:
    print("This cleanup code is always called")
```

Exception Handling

- จะเห็นว่าจะไม่มี exception error เกิดขึ้นแล้ว
- ในส่วน finally จะถูกเรียกมาทำงานในทุกกรณี ซึ่งจะมีประโยชน์ในการทำงานส่วนที่เหลือ เช่น
  - จัดการ database connection ที่ค้างไว้
  - ปิดไฟล์
  - ส่งข้อมูลในระบบเครือข่ายเพื่อแจ้งข้อมูล

Exception Handling

- จะสังเกตได้ว่าทุก exception ก็เป็น object
- จะเป็น Subclass ของ BaseException และ Exception อีกที
- เนื่องจาก BaseException จะรวม system exit และ KB interrupt ดังนั้นจึงไม่ควรใช้ except เฉยๆ แต่ให้ใช้เป็น except Exception



Our Exception

- เราสามารถกำหนด exception ของเราเองได้

```
class InvalidWithdrawal(Exception):
    def __init__(self, balance, amount):
        super().__init__("account doesn't have ${}".format(amount))
        self.amount = amount
        self.balance = balance

    def overage(self):
        return self.amount - self.balance

try:
    raise InvalidWithdrawal(25, 50)
except InvalidWithdrawal as e:
    print("I'm sorry, but your withdrawal is more than your balance"
          " by ${}".format(e.overage()))
```

Exception Handling

- สรุปว่า exception เป็นวิธีการในการจัดการกับ error วิธีหนึ่ง
- อาจจะใช้การตรวจสอบเองก็ได้ตามตัวอย่าง

```
def divide_with_exception(number, divisor):
    try:
        print("{} / {} = {}".format(number, divisor, number / divisor * 1.0))
    except ZeroDivisionError:
        print("You can't divide by zero")

def divide_with_if(number, divisor):
    if divisor == 0:
        print("You can't divide by zero")
    else:
        print("{} / {} = {}".format(number, divisor, number / divisor * 1.0))
```

Exception Handling

- สำหรับใน Class ก็สามารถนำมาเลือกใช้ได้

```python
def set_items1(self, new_items):
    if isinstance(new_items, int):
        self._items = new_items
    else:
        print("Please enter a valid list of items.")

def set_items2(self, new_items):
    try:
        new_items = int(new_items)
    except ValueError:
        return
    self._items = new_items
```

For your attention