Série TP n°8 2022-2023	Module	MTI Méthd. et tech d'implement.
	Filière	Master GSI 1 ère Année
Chapitre 4: Réflexivité	ملاحظة :	

- **Objectifs** أهداف: Réflexivité
- **Données** بيانات:
- **Outils** أدوات : Python

1 Partie TP

1.1 Fontion type()

Exécuter les scripts suivants et noter les résultats

```
a = 12
b = 13.57
c = True
d = 12 + 3j

print(type(a))
print(type(b))
print(type(c))
print(type(d))
```

```
mystr = 'Salam Alykom!'
mylist = [1, 2, 3, 4, 5, 6, 7]
mytuple = (80, 'TD', 15, 'Cours')

print(type(mystr))
print(type(mylist))
print(type(mytuple))
```

```
my_tuple = (10, 'Hello', 45, 'Hi')
my_dict = {1: 'One', 2: 'Two', 3: 'Three'}

if type(my_tuple) is not type(my_dict):
      print("Both variables have different object types.")
else:
      print("Same Object type")
```

La fonction type() permet de créer un type dynamiquement, exécuter le script.
Ensuite explorer la documentation pour les fonctions type() et vars()

```
a = type('Test', (object, ), dict(x = 'Hello', y = 10, z = 'World'))

print(type(a))
print(vars(a))
```

Exemple d'application : calculer en contrôlant le type des arguments

```python
def calculate(x, y, op='sum'):
    if not(isinstance(x, int) and isinstance(y, int)):
        print(f'Invalid Types of Arguments - x:{type(x)}, y:{type(y)}')
        raise TypeError('Incompatible types of arguments, must be integers')
    if op == 'diff':
        return x - y
    if op == 'mult':
        return x * y
    # default is sum
    return x + y
```

Améliorer la fonction calculate pour permettre de faire les calcules avec les entiers et les réels.

1.2 Fontion isinstance()

Tester si les variables sont des instances des types standards.

```python
int_1 = 7
str_1 = "Learn Python"
list_3 = [2, 4, 6]

print ("Is int_1 an integer?" + str (isinstance (int_1, int)))
print ("Is int_1 a string?" + str (isinstance (int_1, str)))
print ("Is str_1 a string?" + str (isinstance (str_1, str)))
print ("Is list_1 an integer?" + str (isinstance (list_1, int)))
print ("Is list_1 a list?" + str (isinstance (list_1, list)))

# test if its types is in a list
print ("Is int_1 integer or list or string?" + str (isinstance (int_1, (list, str, int))))
print ("Is list_1 string or tuple?" + str (isinstance (list_1, (str, tuple))))
```

Tester si une variable est une instance d'une classe

```python
class Test1:
     a = 6

testInstance = Test1 ()

print (isinstance (testInstance, Test1))
print (isinstance (testInstance, (list, tuple)))
print (isinstance (testInstance, (list, tuple, Test1)))
```

1.3 Fonction vars()

```python
class Student:
    def __init__(self, name = 'Leo', age = 22, course = 'MBA', city = 'Mumbai'):
        self.Name = name
        self.Age = age
        self.Course = course
        self.City = city

obj = Student()
print('Dictionary output is:', vars(obj))

obj2 =obj = Student("Samir", 22, "MBB", "Bouira")
print('Dictionary output is:', vars(obj2))
```

1.4 Fonction dir()

Afficher les méthodes et les attributs d'un module ou une classe

```
import random
print("The random library's contents are as follows::")
print(dir(random))
```

Afficher les attributs/méthodes d'une variable de type donné

```
number = [1, 2, 3]
print(dir(number))
```

Re-tester le script prétendant avec une variable de type list, dict, float, et sans paramétré.
Afficher les attributs/méthodes d'une classe.

```
class Student():
     def __init__(self,x):
          return self.x
# Calling function
att = dir(Student)
print(att)
```

On peut personnaliser la méthode dir() pour une classe donnée.

```
class Student():
     def __init__(self,x):
          return self.x
     def __dir__(self):
        return [10,20,30]
s = Student()
att = dir(s)
print(att)
```

1.5 Extraction des détails des classes Python

Analyser le résultat d'exécution de script suivant:

```
class Data:
    """Data Class"""
    d_id = 10


class SubData(Data):
    """SubData Class"""
    sd_id = 20

print(Data.__class__)
print(Data.__bases__)
print(Data.__dict__)
print(Data.__doc__)

print(SubData.__class__)
print(SubData.__bases__)
print(SubData.__dict__)
print(SubData.__doc__)
```

1.6 Modification Dynamique des attributs

```
class A:
    pass
# ajouter dynamiquement un attribut à la classe A
A.x = 1
a = A()

# ajouter dynamiquement un attribut à la l'objet a
a.y = 2

print("Attributs de la classe A: ",vars(A))
print("Attributs de l'instance a: ",vars(a))
```

Étant donné que les méthodes ne sont qu'un type spécial d'attribut, cela signifie que nous pouvons également ajouter des méthodes au moment de l'exécution. Modifions notre classe en y ajoutant dynamiquement une méthode ___init___.

```
def init(self):    # the function and argument can have any name
    self.x = 1
class A:
    pass
setattr(A, '__init__', init)
a = A()
print(a.x)
```

Nous pouvons pousser ce concept un peu plus loin en modifiant l'attribut ___code___ d'une fonction. Cette fois par simple affectation :

```
def test():
    return "Test old code"
print(test())
# ~ test.__code__ = (lambda x : print("Hello", x)).__code__
test.__code__ = (lambda: "Test new code").__code__
print(test())
```

1.7 Methode ___call___()

Python a un ensemble de méthodes intégrées et ___call___est l'une d'entre elles. La méthode ___call___ permet aux programmeurs Python d'écrire des classes où les instances se comportent comme des fonctions et peuvent être appelées comme une fonction. Lorsque l'instance est appelée en tant que fonction ; si cette méthode est définie, x(arg1, arg2, ...) est un raccourci pour x.___call___(arg1, arg2, ...).

```
class Product:
    def __init__(self):
        print("Instance Created")

    # Defining __call__ method
    def __call__(self, a, b):
        print(a * b)

# Instance created
ans = Product()

# __call__ method will be called
ans(10, 20)
```

La fonction callable(): tester si une classe ou une variable peut être appeler comme une fonction

```
print("Is str callable? ", callable(str)) # str class
print("Is len callable? ", callable(len)) # len function
print("Is list callable? ", callable(list)) # list class

num=10
print("Is variable callable? ", callable(num))
```

Tester le script suivant:

```
class student:
    def greet(self):
        print("Hello there")

std = student()
print("Is student class callable? ",callable(student))
print("Is student.greet() callable? ",callable(std.greet))
print("Is student instance callable? ",callable(std))
```

Tester le code suivant:

```
class student:
    def greet(self):
        print("Hello there")

    def __call__(self):
        print("Hello, I am a student.")

std = student()
print("Is student instance callable? ", callable(std))
print(std())
```

2 Application

On veut appliquer ces notions sur un exemple de classe pour la gestion des nombres complexes.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
class ComplexIt(object):
    def __init__(self, real, imag=0.0):
        self.real = real
        self.imag = imag

    def __add__(self, other):
        return ComplexIt(self.real + other.real,
                         self.imag + other.imag)

    def __sub__(self, other):
        return ComplexIt(self.real - other.real,
                         self.imag - other.imag)

    def __mul__(self, other):
        return ComplexIt(self.real*other.real - self.imag*other.imag,
                         self.imag*other.real + self.real*other.imag)
```

```
    def __div__(self, other):
        sr, si, orl, oi = self.real, self.imag, other.real, other.imag # short forms
        r = float(orl**2 + oi**2)
        return ComplexIt((sr*orl+si*oi)/r, (si*orl-sr*oi)/r)

    def __abs__(self):
        return sqrt(self.real**2 + self.imag**2)

    def __neg__(self):   # defines -c (c is ComplexIt)
        return ComplexIt(-self.real, -self.imag)

    def __eq__(self, other):
        return self.real == other.real and self.imag == other.imag

    def __ne__(self, other):
        return not self.__eq__(other)

    def __str__(self):
        return '(%g, %g)' % (self.real, self.imag)

    def __repr__(self):
        return 'ComplexIt' + str(self)

    def __pow__(self, power):
        raise NotImplementedError ('self**power is not yet impl. for ComplexIt')
    def _illegal(self, op):
        print 'illegal operation "%s" for ComplexIt numbers' % op
    def __gt__(self, other):  self._illegal('>')
    def __ge__(self, other):  self._illegal('>=')
    def __lt__(self, other):  self._illegal('<')
    def __le__(self, other):  self._illegal('<=')



if __name__ == '__main__':

    a = ComplexIt(1,5)
    b = ComplexIt(3,2)

    # usual operations
    print(a)
    print(b)
    print(a+b)
    print(a-b)
    print(a*b)
    print(a/b)
    print(a<=b)

    # addition with integer
    x=  a + 4.5
    y=  a + ComplexIt(4.5, 0)
    print(x, y, x==y)
```

3 Travail à domicile

واجب منزلي

لا واجب منزلي

4 Références

مراجع

- reflection in Python https://www.geeksforgeeks.org/reflection-in-python/
- Python Reflection and Introspection https://betterprogramming.pub/python-reflection-and-introspectio
- Python Type() https://www.toppr.com/guides/python-guide/references/methods-and-functions/methods built-in/type/python-type/
- Python type() Function [With Easy Examples] https://www.digitalocean.com/community/tutorials/ python-type
- Hans Petter Langtangen, Introduction to classes, 2016 http://hplgit.github.io/primer.html/doc/pub/ class/
- Hans Petter Langtangen, A Primer on Scientific Programming with Python, Springer, 2014