CONCEPTS OBJET

Gaël Roustan, Argonaultes

 gael.roustan

 gael@argonaultes.fr

 argonaultes

 gael-roustan

Système de notation

QCM de 20 à 30 questions

TP Noté à rendre sur Github ou Gitlab

PYTHON

Installer environnement exécution Python

Télécharger l'installation de Python

Bases de programmation en Python

Ecrire un fichier avec l'extension **py**

Exécuter ce fichier avec la commande

```
python file.py
```

Lancer le programme python en interactif

```
python
```

Afficher un message sur la console

```
print('python')
```

Saisir un message depuis la console

```
mymessage = input('type your message')
```

Les principaux types en python

- Chaines de caractères
- Nombres (entiers)
- Nombres à virgules
- Booléens

Utilisation des variables

Permet de conserver une valeur dans le temps

```
mystr = "program"
print(myvar)
myint = 12
print(myint)
```

Opérations entre les nombres

```
>>> 12 + 3 # addition
15
>>> 12 - 3 # soustraction
9
>>> 12 / 3 # division
4.0
>>> 12 / 5
2.4
>>> 12 * 5 # multiplication
60
>>> 12 // 7 # quotient de la division euclidienne
1
>>> 12 % 7 # reste de la division euclidienne
5
>>> 12 ** 3 # puissance
```

AUTRES OPÉRATIONS MATHÉMATIQUES

```python
# On importe l'ensemble de la bibliothèque math dans le progra
from math import *
print(sqrt(3))

# On importe juste la fonction nécessaire
from math import sqrt
print(sqrt(3))

# On importe l'intégralité de la bibliothèque
# mais on doit préfixer la fonction par le nom de la bibliothè
import math
print(math.sqrt(3))
```

EXERCICE

Afficher le message de bienvenue : "Bienvenue ici"

Puis afficher un message demandant de saisir le nombre de points de vie

Stocker cette valeur saisie par l'utilisateur dans une variable

Afficher un nouveau message demandant de saisir le nombre de dégâts reçus

Stocker cette valeur saisie par l'utilisateur dans une

autre variable

Afficher ensuite à l'utilisateur le nombre de points de vie restants

STRINGS

Chaines de caractères indexables

```
word = 'Python'
word[0]  # character in position 0

word[5]  # character in position 5
```

Les indexes peuvent être négatifs

```
word[-1]  # last character

word[-2]  # second-last character

word[-6]
```

Pour les nombres négatifs, le 1er indice commence à -1

STRINGS

Utilisation possible des slices (intervalles)

```
word[0:2]  # characters from position 0 (included) to 2 (exclu

word[2:5]  # characters from position 2 (included) to 5 (exclu
```

La borne supérieure est toujours exclue, ce qui permet par exemple d'avoir les expressions suivantes toujours égales sans chevauchement.

```
word[:2] + word[2:]

word[:4] + word[4:]
```

Voici la correspondance des indexs positifs et négatifs

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

INDEXS TROP GRANDS

Si un index est trop grand par rapport à la taille de la chaine de caractère, une erreur est générée

```
word[42]
# the word only has 6 characters
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Pour les slices, l'index trop grand est utilisable sans erreur

```
word[4:42]

word[42:]
```

IMMUTABILITÉ DES STRINGS

Les chaînes de caractères sont immutables c'est à dire non modifiables. Impossible de changer une lettre sans passer par la création d'une nouvelle chaîne.

F STRINGS, INTERPOLATION ET FORMATTAGE

- Interpolation de valeurs en utilisant les f-strings
- formater les f-strings en utilisant le mini langage de formattage
- voir quelques fonctionnalités principales à partir de Python 3.12
- savoir quand est-ce qu'il est nécessaire d'utiliser les outils traditionnels plutôt que la f-string

https://realpython.com/python-f-strings/

STRUCTURES DE DONNÉES

LISTES

- Une collection ordonnée d'éléments de différents types

```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple'
fruits.count('apple')

fruits.count('tangerine')

fruits.index('banana')

fruits.index('banana', 4)  # Prochain élément 'banana' à parti

fruits.reverse()
fruits

fruits.append('grape')
fruits
```

SET

- Un ensemble (set) est une collection non ordonnée sans aucune duplication d'éléments.
- Les sets supportent les opérations mathématiques sur les ensembles comme l'union, l'intersection, la différence et la différence symétrique

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banan
print(basket)                      # show that duplicates have

# create empty set
fruits = set()
# add an item to set
fruits.add('orange')

'orange' in basket                 # fast membership testing

'crabgrass' in basket


# Demonstrate set operations on unique letters from two words
```

PILE (FILO)

First In Last Out

```
 1 stack = [3, 4, 5]
 2 stack.append(6)
 3 stack.append(7)
 4 stack
 5
 6 stack.pop()
 7
 8 stack
 9
10 stack.pop()
11
12 stack.pop()
13
14 stack
```

PILE (FILO)

First In Last Out

```
1  stack = [3, 4, 5]
2  stack.append(6)
3  stack.append(7)
4  stack
5
6  stack.pop()
7
8  stack
9
10 stack.pop()
11
12 stack.pop()
13
14 stack
```

PILE (FILO)

First In Last Out

```
1  stack = [3, 4, 5]
2  stack.append(6)
3  stack.append(7)
4  stack
5
6  stack.pop()
7
8  stack
9
10 stack.pop()
11
12 stack.pop()
13
14 stack
```

PILE (FILO)

First In Last Out

```
1  stack = [3, 4, 5]
2  stack.append(6)
3  stack.append(7)
4  stack
5
6  stack.pop()
7
8  stack
9
10 stack.pop()
11
12 stack.pop()
13
14 stack
```

PILE (FILO)

First In Last Out

```
1  stack = [3, 4, 5]
2  stack.append(6)
3  stack.append(7)
4  stack
5
6  stack.pop()
7
8  stack
9
10 stack.pop()
11
12 stack.pop()
13
14 stack
```

PILE (FILO)

First In Last Out

```
1  stack = [3, 4, 5]
2  stack.append(6)
3  stack.append(7)
4  stack
5
6  stack.pop()
7
8  stack
9
10 stack.pop()
11
12 stack.pop()
13
14 stack
```

PILE (FILO)

First In Last Out

```
 1 stack = [3, 4, 5]
 2 stack.append(6)
 3 stack.append(7)
 4 stack
 5
 6 stack.pop()
 7
 8 stack
 9
10 stack.pop()
11
12 stack.pop()
13
14 stack
```

FILE (FIFO)

First In First Out

L'implémentation d'une file (queue) avec les les listes est couteux car la sortie d'un élément oblige à décaler les indices de tous les éléments restants.

```
from collections import deque
queue = deque(["Eric", "John", "Michael"])
queue.append("Terry")           # Terry arrives
queue.append("Graham")          # Graham arrives
queue.popleft()                 # The first to arrive now leav

queue.popleft()                 # The second to arrive now lea

queue                           # Remaining queue in order of
```

TUPLES

- Un tuple correspond à plusieurs éléments séparés par des virgules.
- Un tuple est immutable.
- Un tuple peut être composé d'autres tuples ou d'autres types comme des listes.

```
t = 12345, 54321, 'hello!'
t[0]
t
# Tuples may be nested:
u = t, (1, 2, 3, 4, 5)
u
# Tuples are immutable:
t[0] = 88888
# but they can contain mutable objects:
v = ([1, 2, 3], [3, 2, 1])
v
```

TUPLE À 0 OU 1 ÉLÉMENT

Pour déclarer un tuple vide ou un tuple à 1 élément, la syntaxe est particulière

```
empty = ()
singleton = 'hello',    # <-- note trailing comma
len(empty)

len(singleton)

singleton
```

EXERCICE TUPLES

EXERCICE TUPLES

Créer un tuple de 3 éléments dont vous pouvez changer la valeur des éléments.

EXERCICE TUPLES

Créer un tuple de 3 éléments dont vous pouvez changer la valeur des éléments.

Aide : L'élément à changer peut être stocké dans une autre structure que l'on vient de voir.

DICTIONNAIRES

- Aussi appelé mémoire associative ou tableau associatif.
- L'index numérique séquentiel est remplacé par une clé qui peut être
  - une chaine de caractère immutable
  - un entier
  - un tuple d'objets immutables

```
tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127
tel

tel['jack']

del tel['sape']
tel['irv'] = 4127
tel

list(tel)

sorted(tel)
```

Pour construire un dictionnaire

```
dict([('shape', 3112), ('guido', 413), ('jack', 8931)])
```

Création d'un dictionnaire via une itération

```
{x: x**2 for x in (2, 4, 6,)}
```

Si les clés sont des simples chaînes de caractères

```
dict(shape=3131, guido=3113, jack=8931)
```

SEQUENCES

Une séquence est une structure de données indexées numériquement : listes, tuples et ensembles.

Sequence packing

```
t = 12345, 54321, 'hello!'
```

Sequence unpacking

```
x, y, z = t
```

ITÉRATIONS

ITÉRATIONS SUR UN DICTIONNAIRE

Pour récupérer les clés en même temps que les valeurs

```
knights = {'robin': 'the brave', 'lancelot': 'the humble'}
for key, value in knights.items():
    print(key, value)
```

ITÉRATIONS SUR UNE SÉQUENCE

Une séquence indexée numériquement est une liste, un set ou un tuple

```
sequence_list=[21, 14, 90]
for index, value in enumerate(sequence_list):
    print(index, value)

sequence_tuple=('item1', 'item2', 'item3')
for index, value in enumerate(sequence_tuple):
    print(index, value)

sequence_set={'item1', 'item1', 'item2', 'item3'}
for index, value in enumerate(sequence_set):
    print(index, value)
```

ITÉRATIONS SUR UNE SÉQUENCE

Pour itérer sur 2 séquences ou plus en même temps, les entrées peuvent être regroupées avec la fonction `zip`

```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
    print('What is your {0}?  It is {1}.'.format(q, a))
```

EXERCICE

- Vous devez représenter des vaisseaux spatiaux qui ont une couleur, un nombre de passagers, un nombre de places disponibles et un nom.
- Demander à l'utilisateur de créer 3 vaisseaux.
- Une fois la création des vaisseaux terminée, afficher le vaisseau qui a le meilleur taux de remplissage, c'est à dire le plus grand rapport `nombre_de_passagers / nombre_de_places_disponibles`.

ORGANISATION DU CODE

FONCTIONS

Fonction sans paramètre

```
def function():
    print('your function)
```

Fonction avec paramètre

```
def function_wit_param(parameter):
    print(parameter)
```

Fonction avec paramètre par défaut

```
def function_with_default_param(parameter = 'default'):
    print(parameter)
```

Appeler les fonctions

```
function()
function_wit_param()
function_wit_param('parameter')
function_with_default_param()
function_with_default_param('parameter')
```

LIRE UN FICHIER

Fermeture manuelle

```
f = open('read-file.py', 'r')
print(f.read())
f.close()
```

Fermeture automatique

```
with open('read-file.py', 'r') as f:
    print(f.read())
```

ECRIRE DANS UN FICHIER

Fermeture manuelle

```
f = open('test.txt', 'a')
f.write('new line\n')
f.close()
```

Fermeture automatique

```
with open('test.txt', 'a') as f:
    f.write('new line\n')
```

AUTRE DIRECTIVES

Conditions

```
if condition:
    pass
else:
    pass
```

Boucles

```
for item in list:
    pass
```

EXERCICE

Lire un chiffre dans un fichier et indiquer si ce nombre correspond à un étudiant majeur.

Le fichier contient sur chaque ligne uniquement un nombre

```
10
18
12
29
98
17
67
```

CORRECTION

```
file_ages = open('ages.txt', 'r')
for age in file_ages:
    try:
        if int(age) >= 18:
            print('Age {age} est majeur'.format(age = age[:-1]
        else:
            print('Age {age} n\'est pas majeur'.format(age = a
    except:
        print(f'Age {age} non valide')
file_ages.close()
```

EXERCICE 2 LA SUITE DE FIBONACCI

Suite de Fibonacci : Chaque nombre est la somme des 2 précédents.

$$ F_0 = 0, F_1 = 1 $$ et $$ F_n = F_{n-1} + F_{n-2} $$

Implémenter la suite de Fibonacci jusqu'à 10

```
1 a, b = 0, 1
2 while a < 100:
3     print(a)
4     a, b = b, a + b
```

- Multi affectation
- Boucle while sur 1 condition comparant un nombre avec la valeur fournie par une variable
- Indentation du code pour l'inclure dans le corps de la boucle while
- Affichage du nombre en cours
- Réaffectation selon la règle de la suite de Fibonacci

```
1 a, b = 0, 1
2 while a < 100:
3     print(a)
4     a, b = b, a + b
```

- Multi affectation
- Boucle while sur 1 condition comparant un nombre avec la valeur fournie par une variable
- Indentation du code pour l'inclure dans le corps de la boucle while
- Affichage du nombre en cours
- Réaffectation selon la règle de la suite de Fibonacci

```
1 a, b = 0, 1
2 while a < 100:
3     print(a)
4     a, b = b, a + b
```

- Multi affectation
- Boucle while sur 1 condition comparant un nombre avec la valeur fournie par une variable
- Indentation du code pour l'inclure dans le corps de la boucle while
- Affichage du nombre en cours
- Réaffectation selon la règle de la suite de Fibonacci

```
1 a, b = 0, 1
2 while a < 100:
3     print(a)
4     a, b = b, a + b
```

- Multi affectation
- Boucle while sur 1 condition comparant un nombre avec la valeur fournie par une variable
- Indentation du code pour l'inclure dans le corps de la boucle while
- Affichage du nombre en cours
- Réaffectation selon la règle de la suite de Fibonacci

```
1 a, b = 0, 1
2 while a < 100:
3     print(a)
4     a, b = b, a + b
```

- Multi affectation
- Boucle while sur 1 condition comparant un nombre avec la valeur fournie par une variable
- Indentation du code pour l'inclure dans le corps de la boucle while
- Affichage du nombre en cours
- Réaffectation selon la règle de la suite de Fibonacci

LES PRINCIPAUX PARADIGMES

3 paradigmes principaux de programmation

- OOPs
- Programmation Procédurale : série d'instruction séquentielle
- Programmation Fonctionnelle : exclusivement des fonctions

PROGRAMMATION ORIENTÉE OBJET

- Née dans les années 60 (SmallTalk)
- POO : **P**rogrammation **O**rientée **O**bjet

Concepts de la programmation Orientée Objet

La programmation orientée objet découpe le développement en différents morceaux où chacun de ces morceaux suit un schéma particulier.

Chacun de ces morceaux est appelé une **INSTANCE** d'un schéma également défini par le développeur.

Une instance a des attributs et des méthodes.

PRINCIPES

Héritage

Encapsulation

Polymorphisme

Abstraction

Qu'est-ce que la programmation orientée objet ?

Comment les problèmes de la vie réelle peuvent être résolus avec la programmation orientée objet ?

Pourquoi est-il important d'apprendre la programmation orientée objet ?

Limites et avantages de la programmation orientée objet ?

Pourquoi préférer la programmation orientée objet par rapport à la programmation procédurale ?

CRITÈRES POO

- Paradigme de développement qui décompose tout en objet
- Les caractéristiques de ces objets sont des attributs ou propriétés
- Le comportement de ces objets sont fournies via des méthodes ou des fonctions

ORIENTÉ OBJET

- Chaque brique du développement est considérée comme un objet
- Chaque objet a des attributes (qui définit son état)
- Chaque objet a des capacités (méthode ou fonction qui définit ses actions possibles)
- Chaque objet peut être aussi composé d’objets
- Chaque objet peut partager des attributs avec un autre objet

OBJECTIF

- Modélisation objets concrets (une partie du monde physique qui nous entoure)
- Modélisation objets abstraits (des concepts théoriques)

MODÉLISATION D'UNE MAISON

- Une caractéristique d’une maison peut être la “superficie”, le “prix”, le nombre de chambres : ce sont des attributs
- Eléments principaux d’une maison : des pièces
- Un autre objet : la pièce
- Une maison a donc plusieurs pièces, on dit qu’il y a plusieurs instances de l’objet pièce

ENCAPSULATION

- Regrouper au sein d'une même structure et pouvoir masquer le fonctionnement interne de l'objet : **Encapsulation**.

HÉRITAGE

- La chambre parentale partage certaines propriétés d'une chambre standard, la chambre parentale peut donc hériter des propriétés de la chambre tout en ayant le loisir d'en ajouter : **Héritage** de pièces.

OBJET EN PYTHON

- Définition d'une **classe** : plan d'un objet
- Définition d'un **objet** : instance d'une classe

```
class ClassName:
    pass
```

OPÉRATION SUR LES CLASSES OBJETS

- instanciation : création d'un objet à partir d'une classe

```
class MyClass:
    pass

x = MyClass()
```

INSTANCIATION D'UNE CLASSE

Définition possible de constructeurs

```
class MyClass:
    def __init__(self, param):
        self.param = param
```

Exemple avec la création de nombres complexes

```
class Complex:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag
c1 = Complex(12, 9)
c1.real, c1.imag
```

OPÉRATION SUR LES INSTANCES OBJETS

Référence d'attribut

- attribut de type données
- attribut de type méthodes

```
class MyClass:
    i = 19 # class variable

    def display_i(self):
        print(self.i)
```

Il est possible ensuite de réaliser les références suivantes

```
c = MyClass()
print(c.i)
print(c.display_i())
```

ATTRIBUTS

Pas obligatoire de les déclarer au niveau de la classe

```
c = MyClass()
c.other_attr = 'other'
```

UTILISATION CLASSIQUE

```
class Dog:

    kind = 'canine'         # class variable shared by all ins

    def __init__(self, name):
        self.name = name    # instance variable unique to each

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
```

CONTRE EXEMPLE

```
class Dog:

    tricks = []             # mistaken use of a class variable

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

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
```

EXEMPLE CORRIGÉ

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for eac

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
```

ATTRIBUT PRIVÉ

Contrairement à d'autres langages objets, l'attribut privé n'existe pas en Python

L'artifice consiste à préfixer par un double ___

```
def MyClass:
    def __init__(self):
        self.__hidden = ['hello']

c = MyClass()
c.__hidden # error
c._MyClass__hidden # valid
```

HÉRITAGE

Partager attributs et méthodes entre plusieurs classes.

```
class Parent():
    pass

class Child(Parent):
    pass
```

HÉRITAGE AVEC CONSTRUCTEUR

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

    def get_name(self):
        return self.__name

class Child(Parent):
    def __init__(self, name):
        super(Child, self).__init__(name)

john = Child('toto')
print(john.get_name())
```

HÉRITAGE MULTIPLE

En python, il est possible de dériver de plusieurs classes pour une unique classe

```
class Furniture(object):
    def __init__(self):
        print(f'init Furniture')
        super().__init__()

class Table(object):
    def __init__(self):
        print(f'init Table')
        super().__init__()

class Desk(Furniture, Table):
    def __init__(self):
        super().__init__()

desk = Desk()
```

EXERCICE DES VAISSEAUX

Vous devez modéliser une course de vaisseaux spatiaux.

Ces vaisseaux ont comme particularité un nom, une couleur, une vitesse nominale, une durée d'accélération pour atteindre la vitesse nominale.

Les vaisseaux évoluent sur un circuit qui a une distance d'un tour et un nombre de tours.

Pour une configuration donnée, à chaque tick, indiquer la position des vaisseaux.

EXERCICE

- Créer une classe moldu qui a un attribut nom
- Créer une classe sorcier qui a un attribut maison
- Faire en sorte que la classe sorcier hérite de la classe moldu
- Créer une instance de sorcier
- Créer une instance de moldu
- Ajouter une méthode lancer_un_sort à la classe Sorcier.
- La méthode lancer_un_sort() affiche `Lancer un sort par {nom_du_sorcier}`
- Tenter de lancer un sort depuis l'instance du sorcier.
- Tenter de lancer un sort depuis l'instance du

SOLID

Acronyme pour 5 principes de l'orienté objet

A appliquer autant que possible mais pas à tout prix !

S : SINGLE RESPONSABILITY

Un module ne devrait avoir qu'une unique raison de changer

Cela favorise la composition de classes

Evite la duplication de code

Dans l'exemple, chaque classe a sa propre responsabilité : Person représente l'entité PersonDB

permet la sauvegarde de l’entité

Essayer de rassembler des méthodes qui ont la même raison de changer

CONTRE EXEMPLE

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

    def __repr__(self):
        return f'Person(name={self.name})'

    @classmethod
    def save(cls, person):
        print(f'Save the {person} to the database')


if __name__ == '__main__':
    p = Person('John Doe')
    Person.save(p)
```

O : OPENED CLOSED

Ouvert à l'extension, fermé à la modification

Favoriser un code qui permet en cas de modification de comportement de ne pas changer le code existant

CONTRE EXEMPLE

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

    def __repr__(self):
        return f'Person(name={self.name})'


class PersonStorage:
    def save_to_database(self, person):
        print(f'Save the {person} to database')

    def save_to_json(self, person):
        print(f'Save the {person} to a JSON file')
```

L : LISKOV SUBSTITUTION

Un sous-type (enfant) peut être remplacé par son parent

CONTRE EXEMPLE

```python
from abc import ABC, abstractmethod


class Notification(ABC):
    @abstractmethod
    def notify(self, message, email):
        pass


class Email(Notification):
    def notify(self, message, email):
        print(f'Send {message} to {email}')


class SMS(Notification):
```

I : INTERFACE SEGREGATION

Utiliser des interfaces pour déclarer les comportements à adopter Implémenter les intefaces pour décrire les comportements

CONTRE EXEMPLE

```
class Vehicle(ABC):
    @abstractmethod
    def go(self):
        pass

    @abstractmethod
    def fly(self):
        pass

class Aircraft(Vehicle):
    def go(self):
        print("Taxiing")

    def fly(self):
        print("Flying")
```

D : DEPENDENCY INVERSION

Les modules haut-niveau ne doivent pas dépendre des modules bas-niveau Les implémentations doivent dépendre de l'abstraction et pas l'inverse

CONTRE EXEMPLE

```
class FXConverter:
    def convert(self, from_currency, to_currency, amount):
        print(f'{amount} {from_currency} = {amount * 1.2} {to_
        return amount * 1.2


class App:
    def start(self):
        converter = FXConverter()
        converter.convert('EUR', 'USD', 100)


if __name__ == '__main__':
    app = App()
    app.start()
```

PARTICULARITÉS PYTHON

- Pas d'interface au sens strict, pas de mot clé pour déclarer une interface
- Multi-Héritage

HAPPY POO