Python3 Base

Petit Mémo des manipulation:

Couper / diviser une string après chaque espace:

```
string = "This is a string with spaces"

list = string.split(" ")

print(list)
```

Transformer une string en remplaçant un élément:

`string = "The_universal_number_is_42" new_string = string.replace("_", " ") print(new_string)`

Lister les premières valeurs de du dictionnaire avec boucle FOR IN:

def point_5():

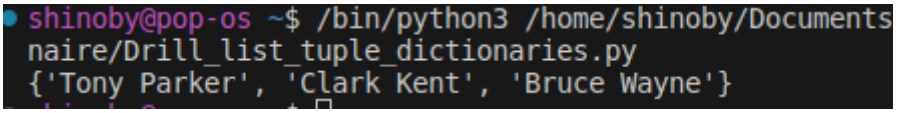
heroes = {"Superman": "Clark Kent", "Batman": "Bruce Wayne", "Spiderman": "Tony Parker"}

values = {hero for hero in heroes.values()}

print(values)

point_5()

Réponce:



Nom des variable:

first_name = "James" # String

last_name = "Bond" # String

age = 39 # Integer

weight = 81.56 # Float

double_agent = True # Boolean

login = "007" # String

agent = [first_name, last_name, age, weight, double_agent, login] # List

print(agent)

Name	Type	Description
Integers	int	Whole numbers, such as : **1, 67, 5000**
Floating point	float	Decimal point numbers, such as : **1.89, 0.67, 9.99999**
Strings	str	Ordered sequence of characters : **"Hello", "10"**
Lists	list	Ordered sequence of objects : **["hello", 10, 56.89]**
Dictionaries	dict	Unordered (Key : Value) pairs : **{"Key1": value, "name" : "Peter}**
Tuples	tuple	Ordered sequence of objects (immutable) : **("hello", 10, 56.89)**
Sets	set	Unordered collections of unique objects : **{1,2}**
Booleans	bool	Logical value : **True or False**

Arithmetic operators

Arithmetic Operators perform various arithmetic calculations like addition, subtraction, multiplication, division, modulus, exponent, etc. There are various methods for arithmetic calculation in Python : you can use the `eval()` function, or declare variables and do the computation, or call functions.

Operator	Description
+ Addition	Adds values on either side of the operator.
- Subtraction	Subtracts right hand operand from left hand operand.
* Multiplication	Multiplies values on either side of the operator.
/ Division	Divides left hand operand by right hand operand.
% Modulus	Divides left hand operand by right hand operand and returns remainder.
** Exponent	Performs exponential (power) calculation on operator.s
// Floor division	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) –

Comparison Operators

These operators compare the values on either side of the operands and determine the relation between them. It is also referred as relational operators. Various comparison operators are (==, != , <>, >,<=, etc)

Operator	Description
==	If the values of two operands are equal, then the condition becomes true.
!=	If values of two operands are not equal, then condition becomes true.
<>	If values of two operands are not equal, then condition becomes true. It has now been removed in Python 3.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.

+= Add AND	Il ajoute l'opérande droit à l'opérande gauche et assigne le résultat à l'opérande gauche.
-= Subtract AND	Il soustrait l'opérande droit de l'opérande gauche et attribue le résultat à l'opérande gauche.

*= Multiply AND	Il multiplie l’opérande droit avec l’opérande gauche et assigne le résultat à l’opérande gauche.
/= Divide AND	Il divise l’opérande gauche avec l’opérande droit et assigne le résultat à l’opérande gauche.
%= Modulus AND	Il prend le module en utilisant deux opérandes et assigne le résultat à l’opérande gauche.
**= Exponent AND	Effectue un calcul exponentiel (puissance) sur les opérateurs et attribue une valeur à l’opérande gauche.
//= Floor Division	Il effectue la division de plancher sur les opérateurs et assigne la valeur à l’opérande gauche.

Assignment

In [ ]:
```
a = 10
name = "Alan Turing"
print(a)
print(name)
```

Add AND (`+=`)

In [ ]:
```
a += 10
name += " is a good mathematician"
print(a)
print(name)
```

❗ **But beware, we can only add variables of the same type**

The variable `a` is of type integer and the variable `name` is of type string.

In [ ]:
```
name += a  # It's the same as `name = name + a`
```

In [ ]:
```
a += name  # It's the same as `a = a + name`
```

Subtract AND (`-=`)

In [ ]:
```
a = 20
a -= 10
print(a)
```

This operator does not work with strings, unlike the `+=` operator.

In [ ]:
```
name -= "Alan Turing"
```

Multiply AND (`*=`)

This operator works with strings.

]:
```
a = 10
a *= 10
print(a)

text = "Alan Turing"
text *= 10
print(text)
```

Divide AND (`/=`)

]:
```
a = 100
a /= 10
print(a)
```

This operator does not work with strings, unlike the `*=` operator. Note that in this case, Python returns a float.

Modulus AND (`=`)

]:
```
a = 100
a %= 3
print(a)
```

This operator does not work with strings.

if

```
age = 23
if age >= 18:
    print("You're an adult!")
```

elif

It is a contraction of the words "else if"

```
age = 17
if age >= 18:
    print("You're an adult!")
elif age == 17:
    print("You are one year short of being an adult.")
```

else

It is possible to give instructions whatever the possible choices with the keyword `else` .

```
age = 16
if age >= 18:
    print("You're an adult!")
elif age == 17:
    print("You are one year short of being an adult.")
else:
    print("Come back later.")
```

Logical operators

Logical operators in Python that are used for conditional statements are true or false. Logical operators in Python are AND, OR and NOT. For logical operators following condition are applied. The "and" and "or" boolean operators allow building complex boolean expressions, for example:

Operator and

```
student1 = "John"
student2 = "Eric"

if student1 == "John" and student2 == "Eric":
    print("You are John AND Eric")
else:
    print("Welcome anonymous")
```

Operator or

```
student1 = "John"
student2 = "Ludovic"

if student1 == "John" or student2 == "Eric":
    print("Your name is either John or Eric.")
else:
    print("Welcome anonymous")
```

Functions that allow cast : `str(), int (), float()` , etc...

```
age = input("How old are you?")
print("string :", age, type(age))

age = int(age)  # Cast from string to integer
print("integer :", age, type(age))

age = float(age)  # Cast from int to float
print("float :", age, type(age))

age = str(age)  # Cast from float to string
print("string :", age, type(age))
```

```
How old are you?27
string : 27 <class 'str'>
integer : 27 <class 'int'>
float : 27.0 <class 'float'>
string : 27.0 <class 'str'>
shinoby@pop-os ~$
```

Lists

Lists are sequences, ordered collections of objects separated by commas. They are declared with the indicating operator `[]` .

```
my_list = ["a", "b", "c", "d"]
```

A value can be accessed by indicating the index.

```
print(my_list[0])
```

We can specify that we only want the first two elements of the list.

```
print(my_list[:2])
```

We can also reverse the list.

```
print(my_list[::-1])
```

Warning, the indexes always start from 0. It's a little disturbing at first, but you have to get used to it. Other examples of what can be done with the lists.

```
my_list = my_list * 5
print(my_list)
```

```
my_list.append("Add Text")
print(my_list)
```

```
my_list.append("Add Text")
print(my_list)
```

You can put any type of data in it.

```
var = "Hello, I am a variable that contains String values."
other_list = [193.45, "Hello", ["Inception list", "Two elements"], 89, var]
print(other_list)
```

You can also merge two lists.

```
x = [1, 2, 3, 4]
y = [4, 5, 1, 0]
x.extend(y)
print(x)

# OR
x = [1, 2, 3, 4]
y = [4, 5, 1, 0]

xy = x + y
print(xy)
```

Tuples

Python offers a type of data called tuple (term meaning "Table UPLEt"), which is quite similar to a list but cannot be modified. Tuples are therefore preferable to lists wherever you want to be sure that the data transmitted is not modified by mistake within a program. In addition, tuples are less "greedy" in terms of system resources (they take less memory space). Tuples are immutable objects *(We will see what this means a little later.)*

```
tuple_example = ("Moriarty", "Sherlock", "Watson")
list_example = ["Moriarty", "Sherlock", "Watson"]
```

tuple VS list

As we can see, when we try to modify the content of a `Tuple` object, we get an error. However, no problem for the list.

```
tuple_example.append("Lestrade")
```

```
list_example.append("Lestrade")
print(list_example)
```

And as we have seen previously, even if we save the `list_example` variable in another variable and make a modification on one of the lists, both lists will be impacted.

```
second_list_example = list_example
second_list_example.append("Hudson")

print(second_list_example)
print(list_example)
```

Dictionaries

The dictionaries or "associative array" is not a sequence but another composite type. They look similar to the lists to some extent (they are editable like them), but the elements we are going to record will not be arranged in an unchanging order. On the other hand, we will be able to access any of them using a specific index called a key, which can be alphabetical, numerical, or even a composite type under certain conditions. Note: Like in a list, the elements stored in a dictionary can be of any type (numerical values, strings, lists, etc.).

In dictionaries, indexes will be strings of characters, unlike lists.

Since the dictionary type is a modifiable type, we can start by creating an empty dictionary and then fill it in little by little. From a syntactic point of view, a dictionary-type data structure is recognized by the fact that its elements are enclosed in a pair of braces. An empty dictionary will therefore be written as `{}`

```
heroes = {}
heroes["Batman"] = "Bruce Wayne"
heroes["Superman"] = "Clark Kent"
print(heroes)
```

As you can see in the line above, a dictionary appears as a series of elements separated by commas (all enclosed between two braces). Each of these elements consists of a pair of objects: an index and a value, separated by a colon.

In a dictionary, indexes are called keys, so elements can be called key-value pairs. You may notice that the order in which the elements appear in the last line does not correspond to the order in which we provided them. This is absolutely irrelevant: we will never try to extract a value from a dictionary using a numerical index. Instead, we will use the keys:

```
print(heroes["Batman"])
print(heroes["Superman"])
```

Here, Batman and Superman are the keys and Bruce Wayne and Clark Kent are the values.

Unlike lists, it is not necessary to use a particular method to add new elements to a dictionary: simply create a new key-value pair.

```
heroes["Spiderman"] = "Peter Parker"
print(heroes)
```

```
def test():
    heroes = {}
    heroes ["Batman"] = "Bruce Wayne"
    heroes ["Superman"] = "Clark kent"
    print(heroes)
    print(heroes["Batman"])
    print(heroes["Superman"])
test()
```

```
shinoby@pop-os ~$ /bin/python3 /home/shinoby/Document
naire/Drill_list_tuple_dictionaries.py
{'Batman': 'Bruce Wayne', 'Superman': 'Clark kent'}
Bruce Wayne
Clark kent
```

```python
def French_English():
    word = {"sword":"épée", "shield":"bouclier", "castle":"château", "beer":"bière", "cheese":
    word[input("Nouveau mots Anglais: ")] = input("Nouveau mots Français: ")
    print(word)
    return
French_English()
```

```
shinoby@pop-os ~$ /bin/python3 /home/shinoby/Documents/BIBLIOTEQUE_DE_ALEXANDRIE/Becode/Python/cyber_sprint/6.Diction
naire/Drill_list_tuple_dictionaries.py
{'Batman': 'Bruce Wayne', 'Superman': 'Clark kent'}
Bruce Wayne
Clark kent
Nouveau mots Français: beurre
Nouveau mots Anglais: butter
{'sword': 'épée', 'shield': 'bouclier', 'castle': 'château', 'beer': 'bière', 'cheese': 'fromage', 'butter': 'beurre'
}
shinoby@pop-os ~$
```