**Python basic**

-         List[] and Tuples(): both contains items but tuples cannot be **UPDATED (read-only).**

***Distinguish between Dictionary{}(dict), List[] and Tuple() in Python***

**Dictionary:**

* Is a data structure that stores data as **key-value pairs**. "key":value
* The key is unique and no duplication is allowed.
* The value can be any data type in Python.
* Access the data in the dictionary using the key.
* Example:

Python

my\_dict = {"name": "Bard", "age": 2, "language": "Python"}

print(my\_dict["name"]) # Output: Bard

**List:**

* Is a data structure that stores data as an **ordered list**.
* List elements can be any data type in Python.
* Access the data in the list by using the index.
* Example:

Python

my\_list = ["a", 1, True, [1, 2, 3]]

print(my\_list[2]) # Output: True

**Tuple:**

* Like a list, a tuple is a data structure that stores data in the form of an **ordered list**.
* However, the tuple **cannot be changed** once created.
* Example:

Python

my\_tuple = ("a", 1, true, [1, 2, 3])

# my\_tuple[2] = False # TypeError: 'tuple' object does not support item assignment

**Comparison table:**

|  |  |  |  |
| --- | --- | --- | --- |
| Feature | Dictionary | List | Tuple |
| Data type | Key-value pairs | Ordered list | Ordered list |
| Change | Subject to change | Subject to change | Can't be changed |
| Access | Use a key | Using indexes | Using indexes |
| Access speed | Fast | Fast | Faster list |
| Use | Store relevant data | Orderly data storage | Store data immutably |

**Usage example:**

* **Dictionary:** Suitable for storing relevant data, e.g. personal information, product information, etc.
* **List:** Suitable for storing ordered data, such as name list, number list, etc.
* **Tuple:** Suitable for storing immutable data, e.g. coordinates, configuration information, etc.

**Conclude:**

Dictionary, list, and tuple are the three basic data structures in Python. The choice of which one to use depends on the intended use and characteristics of the data.

-         Python distinguishes indentation for 1 group of commands

print("Could not find true expression")

print("Good bye!")

#2 The above command goes together

print("trueGood bye!")

***LOOP***

**While loop:**

* **Function:** Repeats a block of code **over and over as long as a certain condition remains true**.
* **Structure:**

Python

while Đ iều\_kiện:

khối\_lệnh

Example:

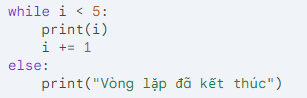
i = 1

while i <= 5:

print(i)

i += 1 # Increase i by 1 after each iteration

         Note that while can still be used with else



**For (for loop) loop:**

* **Function:** Repeats a block of code **for each element in a string, list, tuple or other iterable object**.
* **Structure:**

Python

for biến\_lặp in chuỗi\_hoặc\_danh\_sách:

khối\_lệnh

Example:

fruits = ["apple", "banana", "orange"]

for fruit in fruits:

print(fruit)

**Nested loop**

Here use range(start,end,step)/range(from 0 to [num]) and if to control the "" sign,

For I in range(3):

for j in range(2):

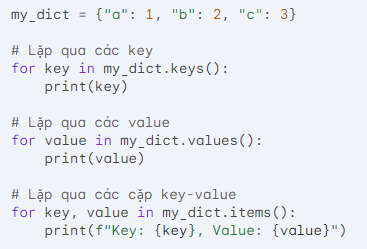
if j < 1:

print(i, j, ",", end=" ")

else:

print(i, j)

**For in dictionary**

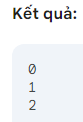


**Break, continue, and pass in Python**

**1. Break:**

* Used to exit the current loop (while, for) or command block.
* When it encounters a break, the program moves on to execute the next instruction after the loop or block of commands.

**Example:**

Python

for i in range(5):

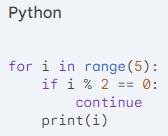
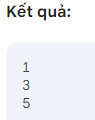
if i == 3:

Break

print(i)

**2. Continue:**

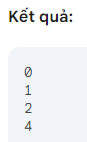
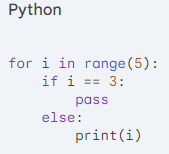
* Use to skip the rest of the current loop and move on to the next one.
* When it encounters continue, the program will not execute the remaining commands in the current loop.

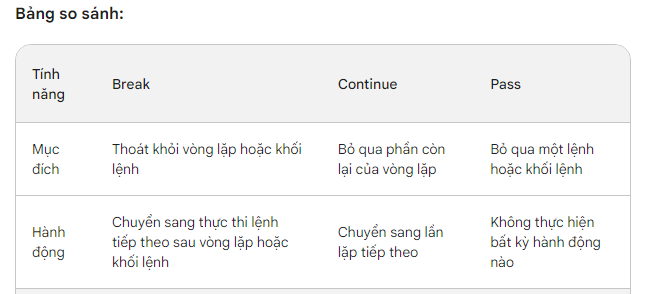
**Example:**

**3. Pass:**

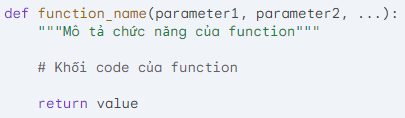
* Used to "ignore" a command or block of commands.
* Pass does not perform any actions, but it can be used to keep the structure of the program intact.

**Example:**





***FUNCTION***



* def: Keywords to declare functions.
* function\_name: The name of the function.
* parameter1, parameter2, ...: The parameters of the function.
* """Describe function function""": Docstring describes function of function (optional).
* Function code block: The commands are executed when calling the function.
* return value: The return value of the function (optional).

! Note: ~~def sum(arg1 = 1, arg2)~~ must be declared for all parameters after the adjective from the first parameter declared

**Python Modules**

Create a new file named [your\_module\_name].py

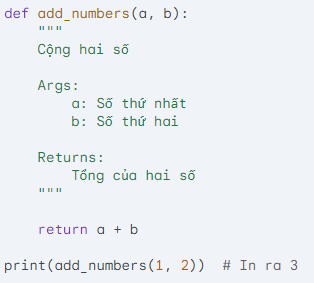
Build functions

When you need sd, import [your\_module\_name].py

Call function from module: [your\_module\_name].function()

**What is Docstring?**

**Docstring** is a string of characters used to describe the functionality of an **object** in Python, including

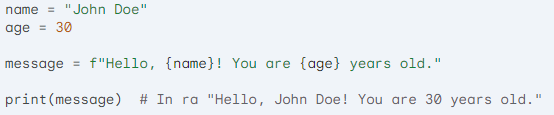


**The + sign can be used to concatenate characters**

print("The youngest child is " + child3)

**In the format string (f-string):**

* Used to **insert expressions** into strings.
* Example:

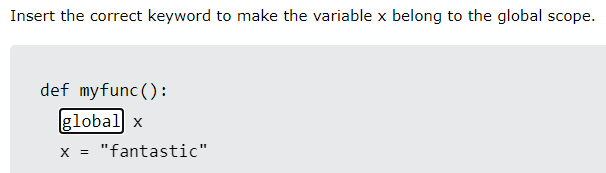


**In the built-in functions:**

* Used to **pass parameters** to some built-in functions.
* Example:

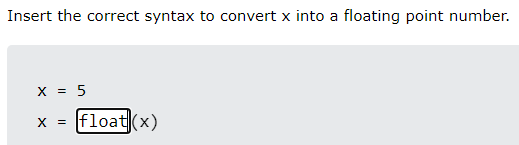


**Global variable**

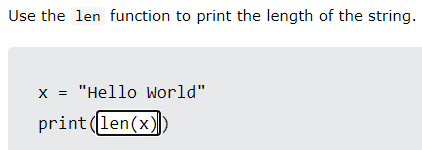


**Data type**: int, str(string), float, list, tuple, dict(dictionary), bool, complex

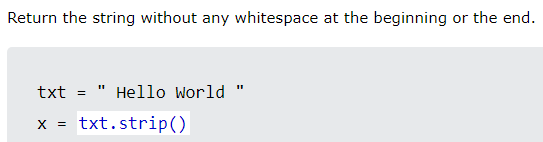
**Convert to another type**



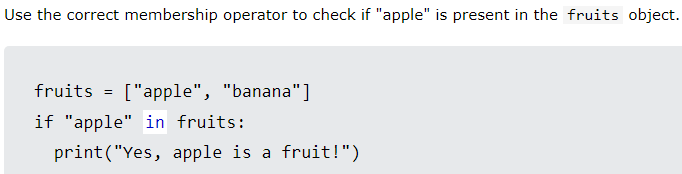
**Length of string**

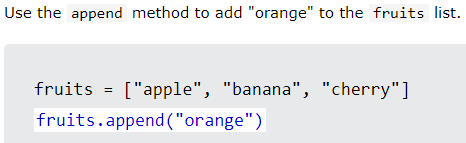


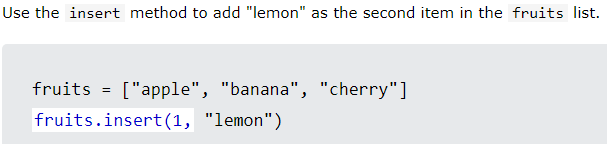
**Remove whitespace**

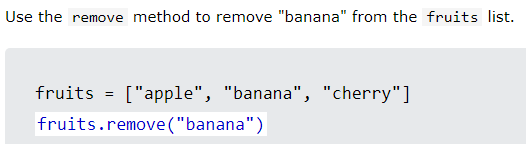


**Check if component available inside dict/tuple/list**

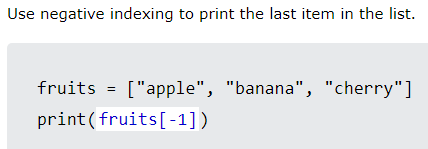


**Add to/remove from list:**

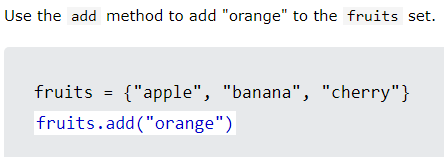
****

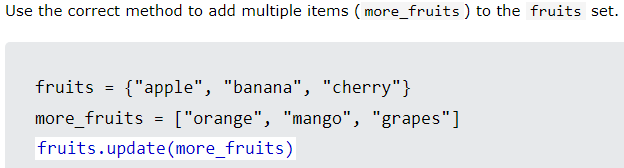
****

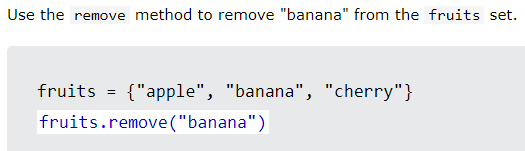
**Print last item using -1 index**

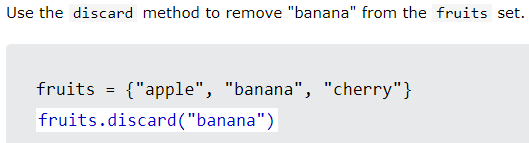
****

**Add to/remove from Sets (single item/list)**

****

****

****

****

**Distinguish discard and remove in Python**

**Both**discard**and**remove**are methods used to remove elements from sets in Python.** However, these two methods have some important differences:

**1. Non-existent element handling:**

* discard**:** If the element to be removed does not exist in the set, the discard method will **not perform any operations** and **will not report an error**.
* remove**:** If the element to be removed does not exist in the set, the remove method will **report a KeyError** error.

**2. Performance:**

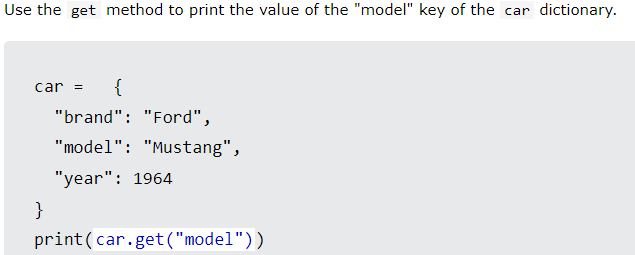
* Discard**:** In general, discard has **higher performance than** remove because it does not need to check if the element exists in the set.
* remove**:** Due to extra checking, removal may be a bit **slower than** discard.

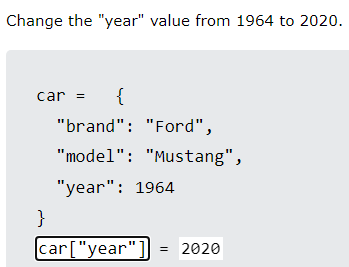
**Conclude:**

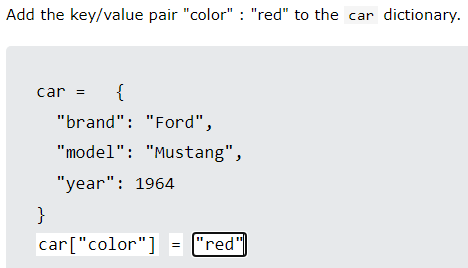
         It is recommended to use discards when you **do not need to care** whether the element exists in the set or not.

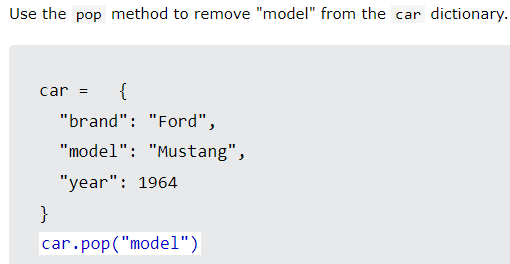
         It is recommended to use remove when you **need to ensure** the element is removed from the set and want to receive an error if the element does not exist.

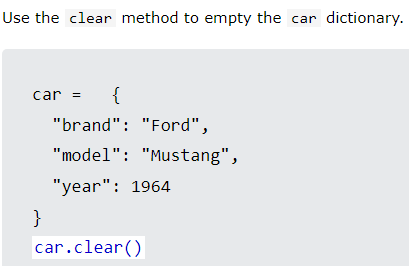
**Dictionary**



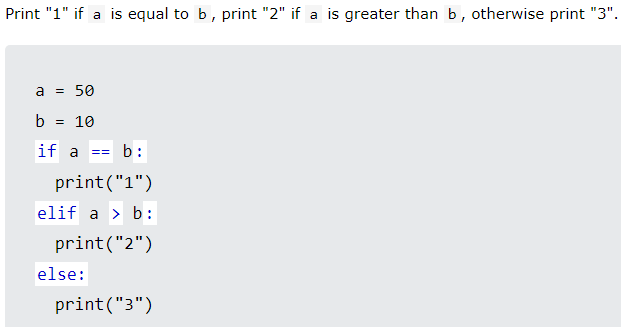


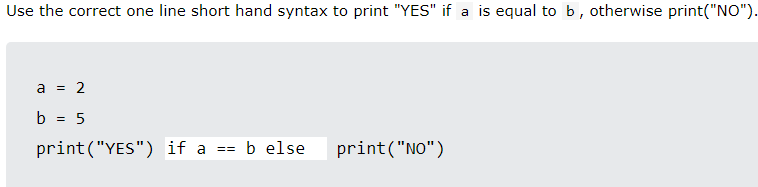




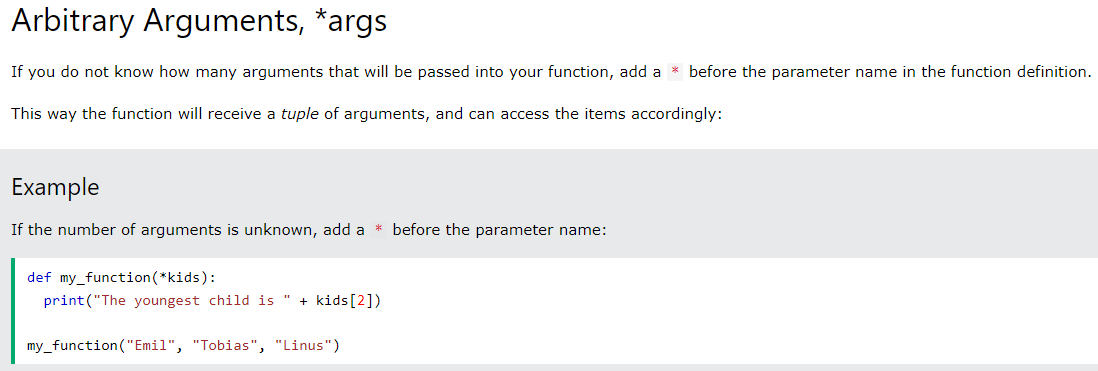


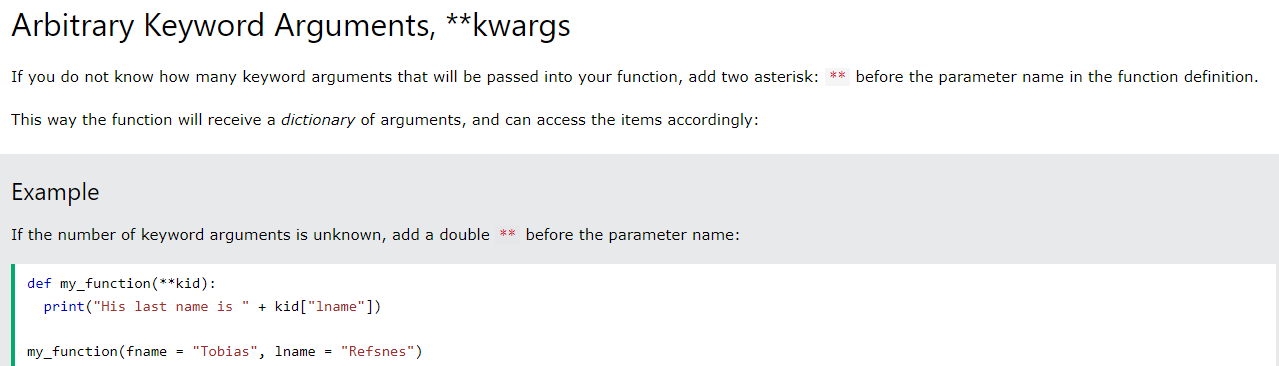
**If…elif…else**





**Function**



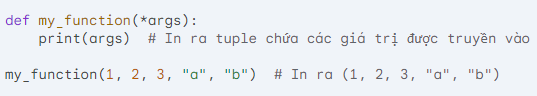


**Meaning of \* and \*\* in function**

**1. Asterisk (\*)**:

**a. Variables of variable length:**

* Used to **collect the remaining values** after declaring the parameters named in the function definition.
* The values are collected into a **tuple**.



**Meaning of \* and \*\* in function**

**1. Asterisk (\*)**:

**a. Variables of variable length:**

* Used to **collect the remaining values** after declaring the parameters named in the function definition.
* The values are collected into a **tuple**.

**Example:**

Python

def my\_function(\*args):

print(args) # Print out the tuple containing the values passed in

my\_function(1, 2, 3, "a", "b") # Print out (1, 2, 3, "a", "b")

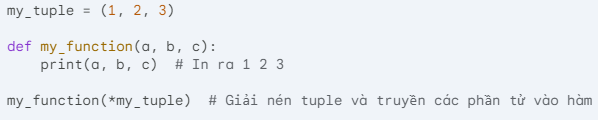
Be [cautious](https://gemini.google.com/faq#coding) when using code snippets.

content\_copy

**b. Unpacking the tuple:**

* Used to **extract elements in the tuple** when passed into the function.

**Example:**



**2. Double asterisk (**):

**a. Variable length keyword variables:**

* Used to **collect the remaining key-value pairs** after declaring the named parameters in the function definition.
* Values collected into a **dictionary**.

**Example:**

def my\_function(\*\*kwargs):

print(kwargs) # Prints to a dictionary containing key-value pairs passed in

my\_function(name="John", age=36, city="Hanoi") # Print out {"name": "John", "age": 36, "city": "Hanoi"}

**b. Extract dictionary:**

* Used to **extract key-value pairs in the dictionary** when passed to the function.

my\_dict = {"name": "John", "age": 36}

def my\_function(name, age):

print(name, age) # Print out John 36

my\_function(\*\*my\_dict) # Unzip the dictionary and pass the key-value pairs to the function

**Conclude:**

* An asterisk (\*) and a double asterisk (\*\*) help define **variables of variable length** in a function.
* Asterisk (\*) for **tuple**, double asterisk (\*\*) for **dictionary**.
* They can also be used to **extract** data structures when passed into functions.

**Lambda function**



**Class**

**The structure of the init function in the class:**

**The init function** is a special method that is called **automatically** when a **new object is created** from the corresponding class. This function is used to **instantiate** the attributes of that object.

**Basic structure:**

Python

def \_\_init\_\_(self, \*args, \*\*kwargs):

# Initialize the properties of the object

self.attribute\_1 = value\_1

self.attribute\_2 = value\_2

    ...

# Handle additional parameters (if any)

if args:

        # ...

if kwargs:

        # ...

**Ingredient:**

1. def \_\_init\_\_(self, \*args, \*\*kwargs)::

o def: Keywords to define methods.

o \_\_init\_\_: The special name of the initialization method.

o self: The first required parameter represents the **newly created object**.

o \*args: **Tuple** parameter to collect the remaining positional arguments **parameters**.

o \*\*kwargs: **Dictionary** parameter to collect the remaining keyword arguments.

2. **Initialize properties:**

o Use self to access and **assign values to object properties**.

3. **Additional parameter handling (optional):**

o Args and kwargs can be used to handle additional parameters passed to the constructor.

**Example:**

Python

Person class:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

person\_1 = Person("John", 36)

person\_2 = Person("Alice", 28)

print(person\_1.name, person\_1.age) # Print out "John 36"

print(person\_2.name, person\_2.age) # Print out "Alice 28"

Be [cautious](https://gemini.google.com/faq#coding) when using code snippets.

content\_copy

**Note:**

* \_\_init\_\_ function **is not required**.
* Without \_\_init\_\_, the properties of the object will **not be initialized**.

**Examples of \*args and \*\*kwargs in class:**

\**1. Examples of args:*

Suppose we have a Car class with a method \_\_init\_\_ get information about the car name and number of seats:

Python

class Car:

def \_\_init\_\_(self, name, seats):

self.name = name

self.seats = seats

car\_1 = Car("Toyota Camry", 5)

car\_2 = Car("Honda Civic", 4)

print(car\_1.name, car\_1.seats) # Print out "Toyota Camry 5"

print(car\_2.name, car\_2.seats) # Print out "Honda Civic 4"

However, if you want to create a car with additional options such as color and engine model, you can use \*args to collect the remaining position parameters:

Python

class Car:

def \_\_init\_\_(self, name, seats, \*options):

self.name = name

self.seats = seats

self.options = options

car\_3 = Car("Tesla Model S", 5, "White", "Electric")

print(car\_3.name, car\_3.seats, car\_3.options) # Print out "Tesla Model S 5 ('White', 'Electric')"

\*\*2. Examples of **kwargs:**

Similarly, we can use \*\*kwargs to collect the remaining named parameters:

Python

Person class:

def \_\_init\_\_(self, name, age, \*\*additional\_info):

self.name = name

self.age = age

self.additional\_info = additional\_info

person\_1 = Person("John", 36, city="Hanoi", occupation="Software Engineer")

print(person\_1.name, person\_1.age, person\_1.additional\_info) # Print out "John 36 {'city': 'Hanoi', 'occupation': 'Software Engineer'}"

\*\*Combination \*args and **kwargs:**

We can combine both \*args and \*\*kwargs in the same method:

Python

MyClass class:

def \_\_init\_\_(self, \*args, \*\*kwargs):

self.args = args

self.kwargs = kwargs

my\_object = MyClass(1, 2, 3, name="John", age=36)

print(my\_object.args, my\_object.kwargs) # Print out((1, 2, 3), {'name': 'John', 'age': 36})

**Note:**

* The order of \*args and \*\*kwargs in the method definition is important.
* When using \*args or \*\*kwargs, be sure to handle the collected values appropriately.

**Conclude:**

args and kwargs are two useful tools that collect parameters passed into methods in Python classes, making coding more flexible and easy.

**Inheritance**

To retain the \_\_init\_\_ of the parent class, use the syntax:  
Student(Person) class:

def \_\_init\_\_(self, fname, lname):

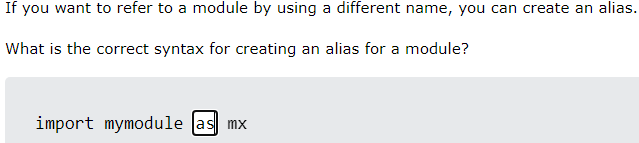
Person.\_\_init\_\_(self, fname, lname) or super().\_\_init\_\_(fname, lname)

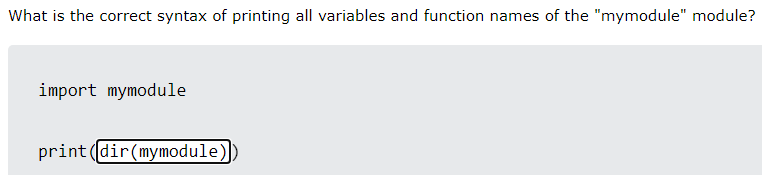
self.graduationyear = year #add properties, same collumn with above line

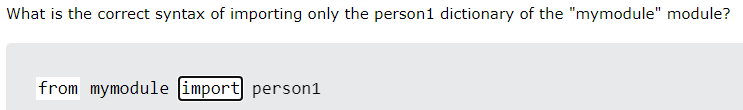
def welcome(self): #add methods

print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)

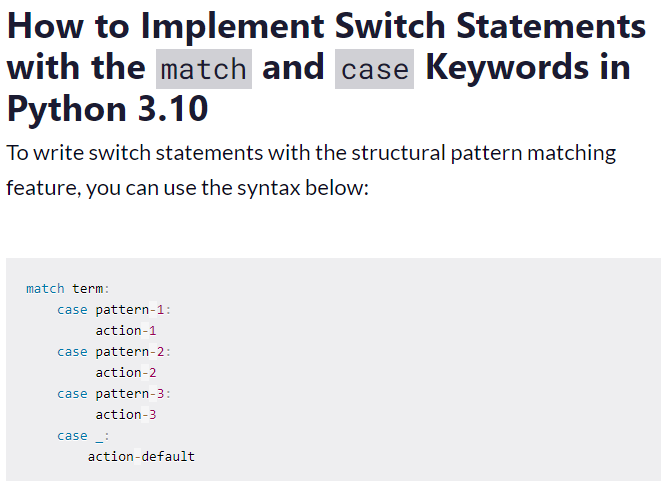
**Module**







**Match – case (switch case replacement)**



**Os, platform, stat and time library**

**1.**platform.system()**:**

This command returns the name of the operating system running the program. For example, on Windows, it will return "Windows".

**2.**platform.uname()**:**

This command returns a tuple containing information about the system, including:

* Operating system name
* Hostname
* Operating system version
* CPU architecture
* Operating system pennant

**3.**os.getcwd()**:**

This command returns the current working directory of the program.

**4.**os.listdir(path)**:**

This command returns a list of files and directories in the directory specified by path.

**5.**os.stat(path)**:**

This command returns an object stat containing information about the file or directory specified by path.

**6.**info.st\_size**:**

* This function returns the size of the file in bytes.
* It indicates how much storage space the file takes up on the device.

**7.**stat.filemode(info.st\_mode)**:**

* This function returns the file's access as a readable string.
* For example, if the result is '-rw-r--r--', that is:
  + - : As a hyphen, there is no specific meaning.
  + r: The owner of the file has read permission.
  + w: The owner of the file has write permissions.
  + -: The group that owns the file does not have permissions.
  + r: Other users (other than owners or groups) have read permissions.
  + -: Other users do not have write permissions.
  + r: Another user has executable permissions (if it is an executable file).

**8.**info.st\_uid**:**

* This function returns the user ID of the file owner.
* It is a unique integer that identifies the user on the system.

**9.**info.st\_dev**:**

* This function returns the device ID where the file is stored.
* It is a unique integer that identifies the storage device on the system.

**10.**time.ctime(info.st\_ctime)**:**

* This function returns the file creation time as an easy-to-read format string (for example, "Thursday, 23 February 2024 09:35:42 +0700").
* info.st\_ctime is the time in seconds since 00:00:00 UTC on January 1, 1970.

**11.**time.ctime(info.st\_mtime)**:**

* This function returns the time the file was last modified as an easy-to-read format string.
* info.st\_mtime is the time in seconds since 00:00:00 UTC on January 1, 1970.

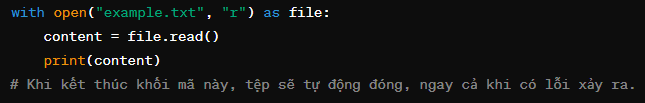
**12.**time.ctime(info.st\_atime)**:**

* This function returns the time the file was last accessed as an easy-to-read format string.
* info.st\_atime is the time in seconds since 00:00:00 UTC on January 1, 1970.

**File**

Use with (to avoid using close) before open("file\_name",'permissions like r, r+, w, w+') as sylas:

Do something



**Split the element to create a new list**

In Python, word[1] for word in words is a structure of list comprehension. List comprehension is a succinct way to create lists in Python.

In this case, words is a list of value sets, and word is a temporary variable that represents each set of values in the list as it iterates through it. word[1] takes the second element in each set of values (since the index in Python starts at 0).

For example, if words are [(1, 'a'), (2, 'b'), (3, 'c')], then word[1] for word in words will generate a list of ['a', 'b', 'c'].

**Regular expression**

<https://www.regular-expressions.info/quickstart.html>

<https://toidicode.com/regular-expression-trong-python-365.html>

<https://toidicode.com/regular-expression-trong-python-p2-366.html>

<https://regex101.com/>

re.match()

The re.search function in Python's re module returns a match object if a regex pattern is found in the supplied string, or None if it is not found.

**The**match object**contains the following information:**

* **Start and end positions of matching strings:** The .start() and .end() attributes provide the start and end positions of the matching string in the original string.
* **Subgroups:** Subgroups in the regex template can be accessed via the .group(), .groups(), and .groupdict() attributes.
* **Entire string match:** The .string attribute provides the entire original string used for the search process.

**Example:**

Python

Import RE

text="Here's an example of using re.search"

# Search for template "re.search"

match = re.search(r"re\.search", text)

if match:

# Print the start and end position of the match sequence

print(f"Start position: {match.start()}")

print(f"End position: {match.end()}")

# Print the entire matching string

print(f"Match string: {match.string}")

else:

print("Regex sample not found")

**Result:**

Start position: 13

End position: 23

Match string: Here's an example of using re.search

**Alternatively, you can use the following flags with the**re.search function**:**

* Re. IGNORECASE**:** Ignores case discrimination when searching.
* **Re. MULTILINE':** Allows ^ and $ to match the beginning and end of each line in the string.
* **Re. DOTALL':** Allows . matches all characters, including new lines.

**Note:**

         The re.search function searches only for the first occurrence of the regex pattern in the string.

         To search for all occurrences of a regex pattern, use the re.findall function.

Hope this explanation helps you better understand how the re.search function works!

**sub(pattern, replace, string, flags)**

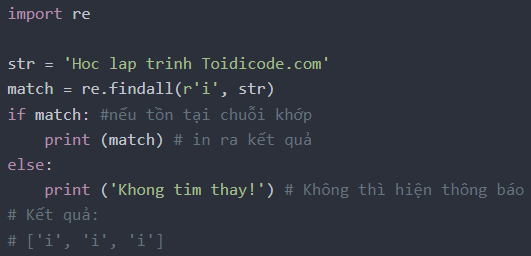
This method works to match and replace matchable strings.

**Where**:

* The pattern is a regular expression string.
* Replace is an alternative string for matched strings.
* String is the string to match.
* flags are the matching types you want to do (see below). If left blank, the flags will be zero.

**findall(partern, string, flags)**

This method has the effect of matching and returns all the strings that it has matched, the rest of the parameters passed and used as for the search method.



* **Special expressions:**
  + \*: Repeat the previous character 0 times or more times
  + +: Repeat the previous character 1 time or more times
  + ?: Repeat the previous character 0 times or 1 time
  + []: Matches any character in the set
  + [^]: Matches any character that is not in the set
  + |: The or sign (or)