

اصول مقدماتی پایتون ۳

functions & modules

هادی فرهادی

شهریور ۱۴۰۴



یادآوری





سؤال جواب





تابع function





تابع یا function



توابع در پایتون بلوکهای کدی هستند که یک کار خاص را انجام میدهند و میتوانند بارها در برنامه فراخوانی - call شوند.



```
def function_name(parameters: type) → return_type:  
    # codes  
    return value # Optional
```

```
def say_hello_to_world():  
    print("Hello, World!")
```



```
# call  
say_hello_to_world()
```





تمرین : تعریف function





پارامتر





پارامتر - positional



```
def greet(name: str, message: str) -> None:
```



```
    """
```

```
    a function that prints message + name
    with positional parameter
    """
```

```
    print(f"{message}, {name}")
```



```
# Call as Positional arguments, order is not important
```

```
greet("Hadi", "Hello")
```

```
greet("Hello", "Hadi")
```

```
# Call as Keyword arguments, order is not important
```

```
greet(message="Hello", name="Hadi")
```

```
# It gives us an error
```

```
greet("Hadi")
```





پارامتر - default



```
def power(first_number: int, second_number: int = 2) -> int:  
    power_result: int = first_number ** second_number  
    return power_result  
    # return first_number ** second_number
```

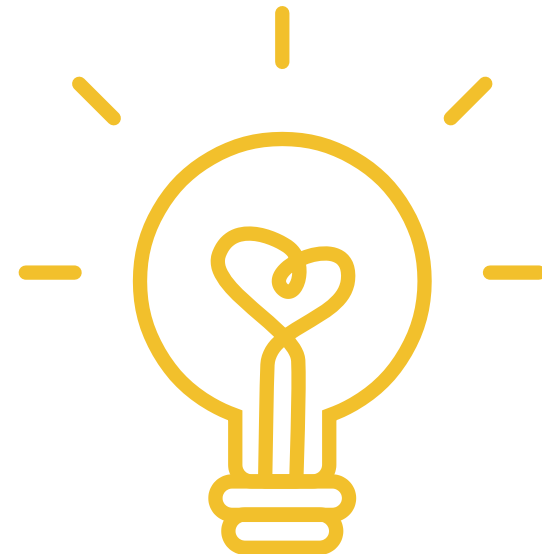


```
result: int = power(10)  
print("power result = ", result)
```



```
result = power(10, 2)  
print("power result = ", result)
```

```
result = power(10, 3)  
print("power result = ", result)
```





نوع tuple





نوع tuple



تاپل یک نوع داده‌ی sequence در پایتون است که شبیه لیست است،



اما با چند تفاوت مهم. تاپل immutable (تغییرناپذیر) است و با پرانتز () تعریف می‌شود.

بعد از انتساب اولیه نمی‌توان مقدار آن را تغییر داد.

سریعتر از لیست است و حافظه‌ی کمتری مصرف می‌کند.

```
empty_tuple: tuple = ()  
print(empty_tuple)
```

```
single_item = (42,) # () is required  
print(single_item) # output: (42,)
```

```
fruits = ("apple", "banana", "cherry")  
numbers = (1, 2, 3, 4, 5)  
mixed = (1, "hello", 3.14, True, [5, 3], (22, 11))
```





نوع tuple

```
fruits = ("apple", "banana", "cherry")  
print(fruits)
```

```
# index  
print(fruits[0])  
print(fruits[-1])
```

```
# Error: changing its value  
fruits[0] = "kiwi"  
print(fruits)
```

```
numbers = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)  
print(numbers[2:5])    # output: (2, 3, 4)  
print(numbers[:3])     # output: (0, 1, 2)  
print(numbers[7:])     # output: (7, 8, 9)  
print(numbers[::-3])   # output: (0, 3, 6, 9)
```

```
print(numbers.count(3))
```

```
print("apple" in fruits) # output: True  
print("orange" in fruits) # output: False
```

```
print(len(fruits)) # output: 3
```

```
print(fruits.index("banana")) # output: 1
```





پارامتر - args*



```
def sum_numbers(*args):
```

```
    """
```

```
    Accepts an unspecified number of Positional Parameters.
```

```
    """
```

```
    print("type of args", type(args))
```

```
    sum_result = 0
```

```
    for number in args:
```

```
        sum_result += number
```

```
    return sum_result
```

```
result = sum_numbers(1, 2, 3, 4, 5, 8)
```

```
print(result)
```



نوشتن تایپ متغیر و کدهای شرکت ها



نوع dict





نوع dict



دیکشنری یک ساختار داده‌ای در پایتون است که داده‌ها را



به صورت جفت‌های کلید-مقدار (key-value) ذخیره می‌کند.

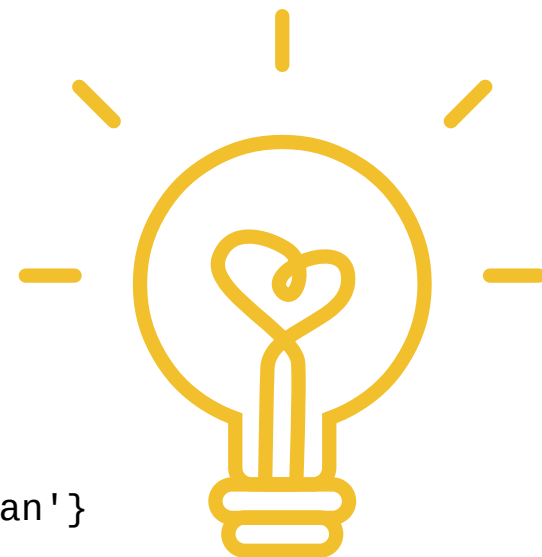
دیکشنری با استفاده از آکولاد {} تعریف می‌شود و هر کلید با یک مقدار مرتبط است.

```
empty_dict: dict = {}  
print(empty_dict) # output: {}
```



```
person: dict = {  
    "name": "Ali",  
    "age": 30,  
    "city": "Tehran"  
}
```

```
print(person) # output: {'name': 'Ali', 'age': 30, 'city': 'Tehran'}
```





نوع dict

```
person: dict[str, any] = {  
    "name": "Ali",  
    "age": 30,  
    "city": "Tehran"  
}
```

```
print(person) # output: {'name': 'Ali', 'age': 30, 'city': 'Tehran'}
```

```
# index
```

```
print(person["name"]) # output: Ali
```

```
print(person.get("name")) # output: Ali
```

```
print(person.get("country", "Iran"))
```

```
# change its value
```

```
person["age"] = 31
```

```
print(person) # output: {'name': 'Ali', 'age': 31, 'city': 'Tehran'}
```

```
# add new item
```

```
person["country"] = "Iran"
```

```
print(person) # output: {'name': 'Ali', 'age': 31, 'city': 'Tehran', 'country': 'Iran'}
```

```
# update value
```

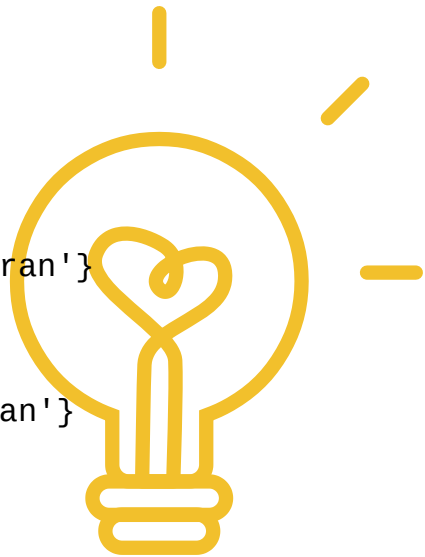
```
person["city"] = "London"
```

```
print(person) # output: {'name': 'Ali', 'age': 31, 'city': 'London', 'country': 'Iran'}
```

```
# remove an item
```

```
del person["city"]
```

```
print(person) # output: {'name': 'Ali', 'age': 31, 'country': 'Iran'}
```





نوع dict

```
# pop an item
age = person.pop("age")
print(age)      # output: 31
print(person)   # output: {'name': 'Ali', 'country': 'Iran'}

# keys
print(person.keys()) # output: dict_keys(['name', 'country'])

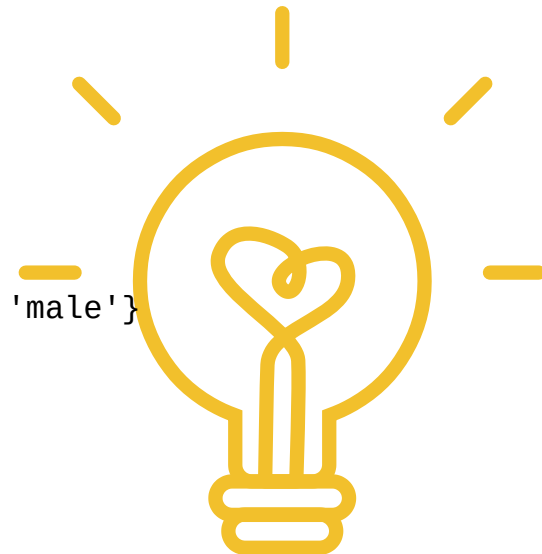
# values
print(person.values()) # output: dict_values(['Ali', 'Iran'])

# items
print(person.items()) # output: dict_items([('name', 'Ali'), ('country', 'Iran')])

# copy without reference
person_copy = person.copy()
print(person_copy) # output: {'name': 'Ali', 'country': 'Iran'}

# update multiple items
person.update({"age": 38, "country": "USA", "sex": "male"})
print(person) # output: {'name': 'Ali', 'age': 38, 'country': 'USA', 'sex': 'male'}

# clear dict
person.clear()
print(person) # output: {}
```





پارامتر - **kwargs



```
def show_fruits(**kwargs):  
    """  
    Accepts an unspecified number of Keyword Parameters.  
    """  
    print(f"Type of kwargs is {type(kwargs)}")  
  
    for key, value in kwargs.items():  
        print(f"{key}: {value}")  
  
show_fruits(apple=3, banana=4, orange=5, mango=13)
```





Positional-Only Parameters - بیشتر بدانید



```
def show_items(first_number, second_number, /, is_admin):  
    print(f"{first_number}, {second_number} and {is_admin}")
```



Correct

```
show_items(1, 3, False)
```

Correct

```
show_items(1, 3, is_admin=False)
```

Not correct

```
show_items(first_number = 1, second_number = 2, is_admin = True)
```



از ابتدا تا علامت / فقط و فقط باید به صورت **positional** کال شود





Positional-Only Parameters - بیشتر بدانید



```
def create_person(name: str, *, age: int, city: str):  
    print(f"{name}, {age}, {city}")
```



Not correct

```
create_person("hadi", 23, "london")
```

Correct

```
create_person(name="hadi", age=23, city="london")
```

Correct

```
create_person("hadi", age=23, city="london")
```



پارامترهای بعد از * باید فقط و فقط به صورت keyword ارسال شوند





ترتیب پارامترها





ترتیب پارامترها



```
def show_values(first_number, second_number, *args, **kwargs):  
    print(first_number, second_number)  
    print(args)  
    print(kwargs)
```

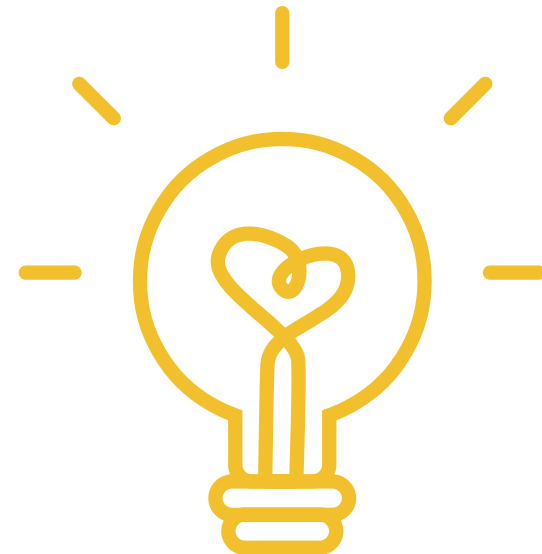


Correct

```
show_values(23, 48, 739, 258, 449, value_1=1, value_2=2)
```

Not Correct

```
show_values(first_number=10, 23, 367, result=2)
```





ترتیب پارامترها



not correct

```
def show_values(first_number=12, second_number):  
    print(first_number, second_number)
```



not correct

```
def show_values(first_number, second_number, **kwargs, *args):  
    print(first_number, second_number)
```





return





return



```
def add(first_number:int, second_number:int) -> int:  
    return first_number + second_number
```

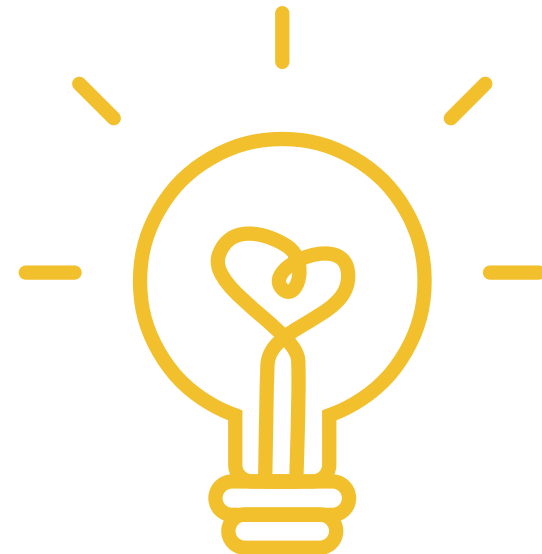


```
result = add(1, 2)  
print(f"result = {result}")
```



```
def absolute_value(number: int):  
    if number >= 0:  
        return number  
    else:  
        return -number
```

```
result: int = absolute_value(-3)  
print(f"result = {result}")
```





return



```
def show_data(first_name: str, last_name: str) -> None:  
    print(f"first_name = {first_name}, last_name = {last_name}")
```

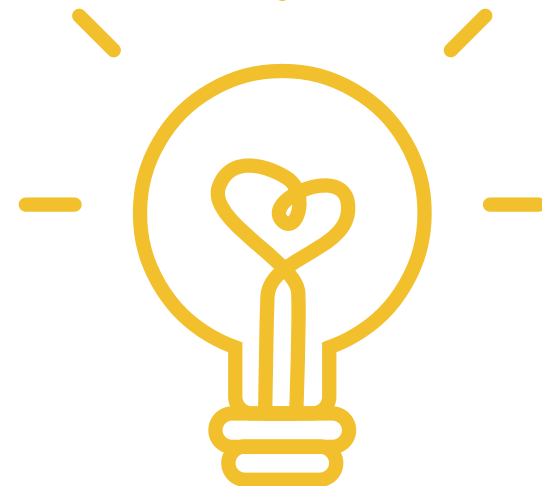


```
def show_data(first_name, last_name):  
    print(f"first_name = {first_name}, last_name = {last_name}")
```



```
def show_data(first_name, last_name):  
    print(f"first_name = {first_name}, last_name = {last_name}")  
    return
```

```
show_data("Alex", "Bob")
```





return



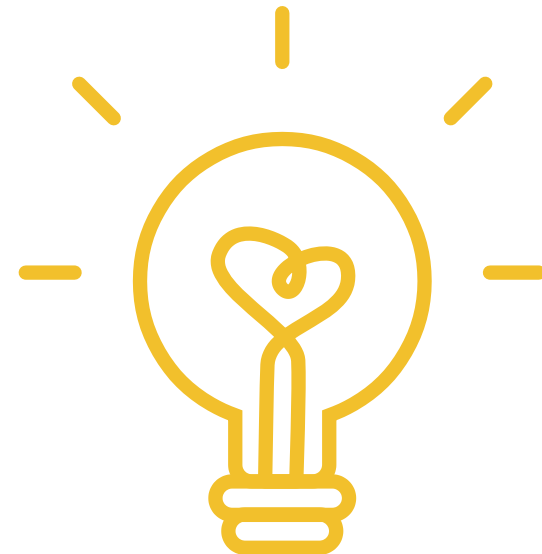
```
def calculate(first_number: int, second_number: int) -> tuple:  
    sum_result = first_number + second_number  
    diff_result = first_number - second_number  
    return sum_result, diff_result
```



```
result = calculate(10, 5) # result = (15, 5)  
sum_result, diff_result = calculate(10, 5) # unpacking: sum=15, diff=5
```



```
print(f"calculation result: {sum_result}")  
print(f"sum result: {sum_result}")  
print(f"diff result: {diff_result}")
```





return



```
def check_age(age: int) -> str:  
    if age < 0:  
        return "Invalid age"
```



```
    if age >= 18:  
        return "Adult"  
    else:  
        return "Minor"
```



```
result = check_age(25) # "Adult"  
print(f"age result: {result}")
```





return



```
def show_info():  
    print("Yes")  
    return "Result"  
    print("No")  # this one won't executed  
  
result = show_info()
```





بیشتر بدانید - Functions as Objects in Python





Functions as Objects in Python - بیشتر بدانید



در پایتون، توابع اشیاء درجه اول (first-class objects) هستند.

این به معنای آن است که توابع را می توان مانند هر شیء دیگری

در پایتون مدیریت کرد، به متغیرها اختصاص داد، به عنوان آرگومان ارسال کرد،

از توابع بازگشت داده شد و در ساختارهای داده ذخیره کرد.



```
def greet(name):  
    return f"Hello, {name}!"  
  
def user_operation(name: str, func: any) -> any:  
    return func(name)  
  
# assign the method to a variable  
my_function = greet  
  
result = user_operation(name="Sara",  
func=my_function)  
  
print(result) # Hello, Sara!
```





بیشتر بدانید - lambda





بیشتر بدانید - lambda



Lambda functions یا توابع بی نام در پایتون، توابع کوچک و یک خطی هستند که می توانند هر تعداد آرگومان بگیرند اما فقط یک عبارت دارند. این توابع با استفاده از کلمه کلیدی lambda ایجاد می شوند.



lambda arguments: expression



```
add = lambda x, y: x + y
```

```
print(add(5, 3)) # output: 8
```

```
numbers = [1, 2, 3, 4, 5]  
squared = list(map(lambda x: x ** 2, numbers))  
print(squared)
```





بیشتر بدانید - lambda



```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)
```



```
words = ["apple", "banana", "cherry", "date"]
sorted_words = sorted(words, key=lambda x: len(x))
print(sorted_words) # output: ['date', 'apple', 'banana', 'cherry']
```



```
pairs = [(1, 5), (3, 2), (2, 8)]
sorted_pairs = sorted(pairs, key=lambda x: x[1])
print(sorted_pairs)
```





scope





scope



```
index = 0
```

```
def increment(count):  
    counter = count
```

```
    index *= 10 # error  
    print(counter)
```

```
increment(10)  
print(counter) # error
```



```
numbers: list = [1, 2, 3]  
squared: list = [number**2 for number in numbers]  
print(number) # error
```





scope



```
for i in range(5):  
    message = f"Hello {i}"
```



```
print(i)    # 4  
print(message)  # Hello 4
```





بیشتر بدانید - scope



```
global_var = 10
```



```
def bad_practice():  
    global global_var  
    global_var += 1
```



```
def better_practice(value):  
    return value + 1
```

```
print(f"global variable: {global_var}")  
global_var = better_practice(global_var)  
print(f"global variable: {global_var}")
```





بیشتر بدانید - scope



```
value = 5 # Don't change this
```



```
def calculate(first_number: int, second_number:int) → int:  
    global value
```

```
    value = first_number + second_number # warning: the value has changed
```

```
    return second_number * second_number * value
```

```
result: int = calculate(3, 4)  
print(value) # warning, value changed
```





بیشتر بدانید - توابع بازگشتی (Recursive Functions)



بیشتر بدانید - توابع بازگشتی (Recursive Functions)

توابع بازگشتی توابعی هستند که خود را فراخوانی می‌کنند تا یک مسئله را به مسائل کوچکتر تقسیم کنند. این توابع برای حل مسائلی که می‌توانند به زیرمسائل مشابه تقسیم شوند بسیار مناسب هستند.

شرط پایه (Base Case): شرطی که بازگشت را متوقف می‌کند

گام بازگشتی (Recursive Step): فراخوانی تابع با ورودی کوچکتر

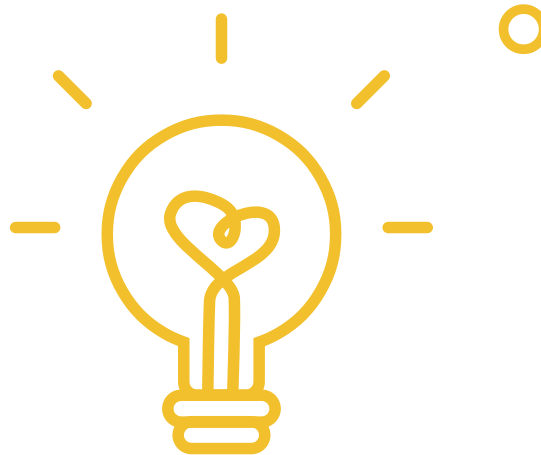
```
def factorial(number: int) -> int:
    # Base Case
    if number == 0 or number == 1:
        return 1
    # Recursive Step
    else:
        return number * factorial(number - 1)

print(factorial(5)) # 120
```





بیشتر بدانید - Call by value, Call by reference



بیشتر بدانید - Immutable Objects (اشیاء تغییرناپذیر)

پس از ایجاد نمی توانند تغییر کنند

اگر سعی در تغییر آنها داشته باشید، یک شیء جدید ایجاد می شود

مثال ها:

اعداد (int, float)

رشته ها (str)

تاپل ها (tuple)

بایت های تغییرناپذیر (bytes)

Boolean (bool)





بیشتر بدانید - Mutable Objects (اشیاء تغییرپذیر)



Mutable Objects (اشیاء تغییرپذیر)

پس از ایجاد می توانند تغییر کنند



تغییرات روی همان شیء اصلی اعمال می شود



مثال ها:

لیست ها (list)

دیکشنری ها (dict)



مجموعه ها (set)

بایت های تغییرپذیر (bytearray)





بیشتر بدانید - Call by Object Reference



پایتون از مدل "Call by Object Reference" استفاده می کند که ترکیبی از Call by Value و Call by Reference است:

برای Immutable Objects: مانند Call by Value عمل می کند



برای Mutable Objects: مانند Call by Reference عمل می کند

```
def process_data mutable_list: list, immutable_param: int):  
    mutable_list.append([66, 77, 99])  
  
    immutable_param += 100
```

```
two_d_list: list = [[1, 2, 3], [3, 4, 5], [7, 8, 9]]  
number: int = 10
```

```
process_data(two_d_list, number)  
print(two_d_list) # the value has changed  
print(number) # it hasn't changed
```





Value type and Reference type - بیشتر بدانید



```
first_str: str = 'Hello World!'
second_str: str = 'Hello World!'
```

```
print(first_str is second_str)
print(id(first_str), id(second_str))
```

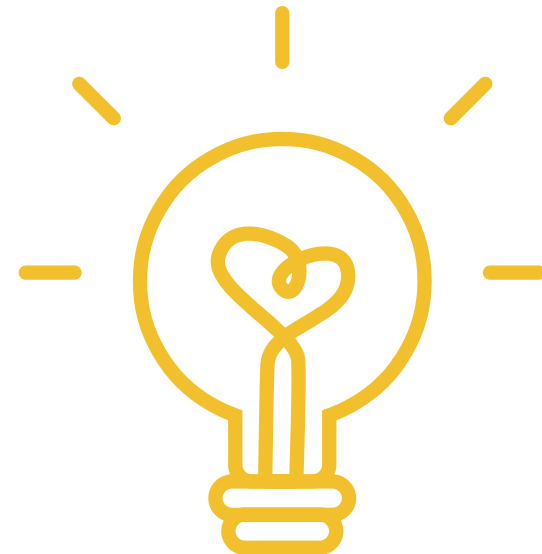
```
first_number_list: list = [1, 2, 3]
second_number_list: list = [1, 2, 3]
third_number_list: list = first_number_list
```

```
print(first_number_list is second_number_list)
print(id(first_number_list), id(second_number_list))
print(third_number_list is first_number_list)
print(id(first_number_list), id(third_number_list))
```

```
first_number_list.append(4)
print(first_number_list) # [1, 2, 3, 4]
print(third_number_list) # [1, 2, 3, 4]
```

```
third_str: str = first_str
first_str += "88"
```

```
print(first_str) # Hello World!88
print(third_str) # Hello World!
```





docstring





Docstring



(مستندسازی) برای توضیح هدف، پارامترها، مقادیر بازگشتی و سایر جزئیات توابع، کلاس ها و ماژول ها استفاده می شود.



```
def connect(host, port, timeout=30):  
    """  
    It connects to the specified host and port  
  
    Args:  
        host (str): Server Address  
        port (int): Server Port  
        timeout (int, optional): Connection Timeout  
    Returns:  
        ServerObject: Server Object  
    """  
    Pass  
print(connect.__doc__)
```





Module





یک ماژول در پایتون فایلی با پسوند py است که شامل کدهای پایتون



(مانند توابع، کلاس‌ها، متغیرها و ...) است. ماژول‌ها به شما اجازه می‌دهند

کدهای خود را سازماندهی کرده و از قابلیت استفاده مجدد (Reusability) بهره ببرید.

کاربردهای اصلی ماژول:

سازماندهی کد: تقسیم کد به بخش‌های منطقی و قابل مدیریت.

قابلیت استفاده مجدد: ذخیره کدها در ماژول‌ها و استفاده از آنها در پروژه‌های مختلف.

پنهان‌سازی اطلاعات: پیاده‌سازی جزئیات داخلی بدون تاثیرگذاری بر دیگر بخش‌های برنامه.





Import Module



```
# math_operations.py
```



```
def add(first_number: int, second_number: int) -> int:  
    return first_number + second_number
```

```
def subtract(first_number: int, second_number: int) -> int:  
    return first_number - second_number
```

```
# main.py
```

```
import math_operations
```

```
sum_result: int = math_operations.add(12, 13)  
print(sum_result)
```

```
subtraction_result: int = math_operations.subtract(12, 13)  
print(subtraction_result)
```





Import Module



```
# math_operations.py
```

```
def add(first_number: int, second_number: int) -> int:  
    return first_number + second_number
```

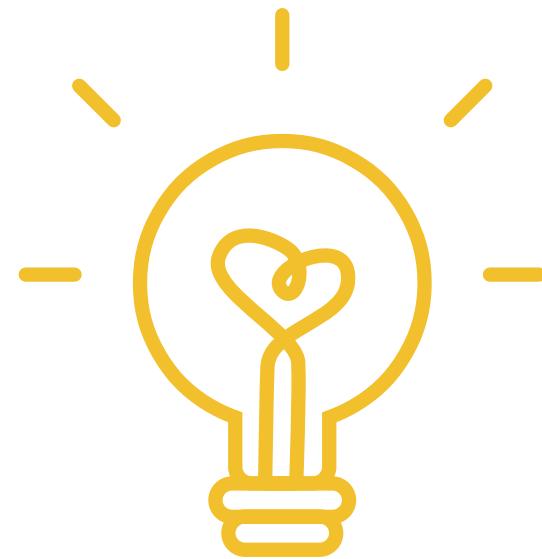
```
def subtract(first_number: int, second_number: int) -> int:  
    return first_number - second_number
```

```
# main.py
```

```
from math_operations import add, subtract
```

```
sum_result: int = add(12, 13)  
print(sum_result)
```

```
subtraction_result: int = subtract(12, 13)  
print(subtraction_result)
```





Import Module



```
# math_operations.py
```

```
def add(first_number: int, second_number: int) -> int:  
    return first_number + second_number
```

```
def subtract(first_number: int, second_number: int) -> int:  
    return first_number - second_number
```

```
# main.py
```

```
# It's not a good way, It has performance issue
```

```
from math_operations import *
```

```
sum_result: int = add(12, 13)  
print(sum_result)
```

```
subtraction_result: int = subtract(12, 13)  
print(subtraction_result)
```





Import Module



```
# math_operations.py
```

```
def add(first_number: int, second_number: int) -> int:  
    return first_number + second_number
```

```
def subtract(first_number: int, second_number: int) -> int:  
    return first_number - second_number
```

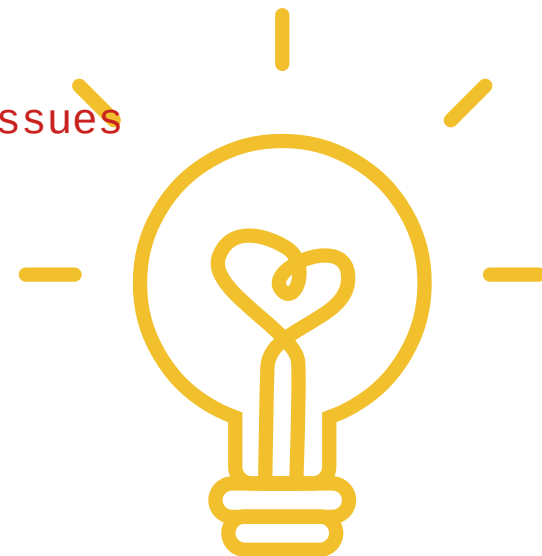
```
# main.py
```

It's not a good way, It has performance readability issues

```
from math_operations import *
```

```
sum_result: int = add(12, 13)  
print(sum_result)
```

```
subtraction_result: int = subtract(12, 13)  
print(subtraction_result)
```





Import Module



math_operations.py

```
def add(first_number: int, second_number: int) -> int:  
    return first_number + second_number
```



```
def subtract(first_number: int, second_number: int) -> int:  
    return first_number - second_number
```



mathematic.py

```
def add(first_number: int, second_number: int) -> int:  
    print(first_number, second_number)  
    return first_number + second_number
```

```
def subtract(first_number: int, second_number: int) -> int:  
    print(first_number, second_number)  
    return first_number - second_number
```





Import Module



```
# main.py
# 3rd place priority
from math_operations import add as math_add, subtract as math_sub
# 2nd place priority
from mathematic import add as mathematic_add, subtract as mathematic_sub
```



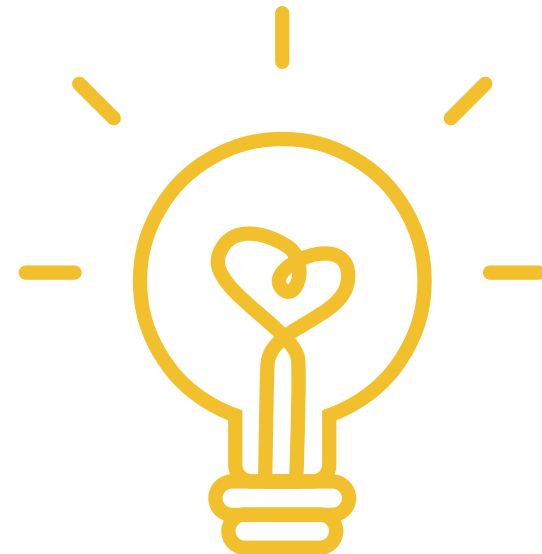
```
def add(first_number: int, second_number: int) -> int: # Highest priority
    return first_number + second_number
```

```
sum_result: int = add(12, 13)
print(sum_result)
```

```
subtraction_result: int = mathematic_sub(12, 13)
print(subtraction_result)
```



```
# Error
subtraction_result: int = subtract(12, 13)
print(subtraction_result)
```





ماژول‌های استاندارد پایتون





ماژول‌های استاندارد پایتون



پایتون دارای کتابخانه استاندارد غنی از ماژول‌های از پیش ساخته شده است مانند:



`math`: برای عملیات ریاضی.

`datetime`: برای کار با تاریخ و زمان.

`os`: برای تعامل با سیستم عامل.

`re`: برای کار با `regex`.





ماژول‌های استاندارد پایتون



math



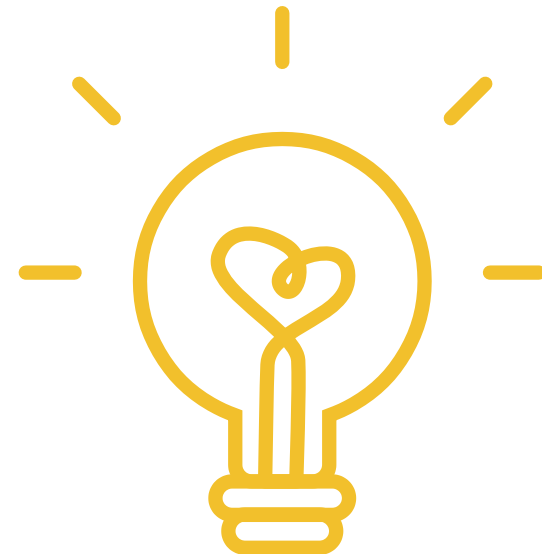
```
from math import sqrt
```



```
number: int = int(input("Enter a number: "))
```

```
result: float = sqrt(number)
```

```
print(f"The square root of {number} is {result}")
```





ماژول‌های استاندارد پایتون



datetime



```
import datetime
```

```
birthday = datetime.date.today()  
print(f"Birthday: {birthday}")
```





بیشتر بدانید - ماژول های استاندارد پایتون



re



```
import re as regular_expression

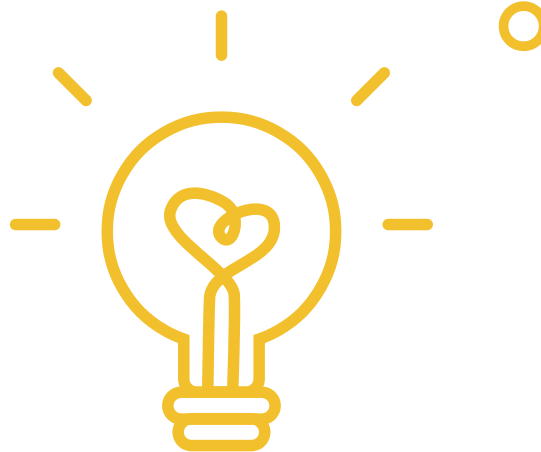
name: str = input("What is your name? ")

if regular_expression.match(r"^H\w+", name):
    print(f"{name} is a great person.")
else:
    print(f"{name} is not a great person.")
```





مسیر جستجوی ماژول‌ها در پایتون





مسیر جستجوی ماژول‌ها در پایتون



۱. دایرکتوری فعلی



اولین جایی که پایتون جستجو می‌کند، دایرکتوری‌ای است که اسکریپت اصلی در آن قرار دارد.

۲. مسیرهای موجود در متغیر محیطی PYTHONPATH



مسیرهای تعریف شده در متغیر محیطی PYTHONPATH (در صورت وجود).

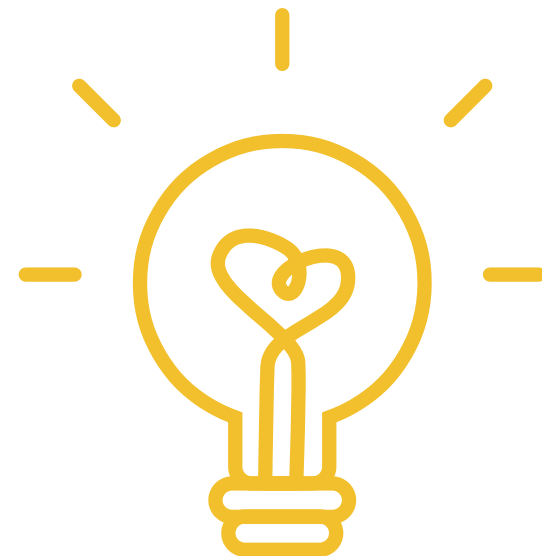
۳. مسیرهای استاندارد پایتون

مسیرهای پیش‌فرضی که پایتون نصب شده است، شامل ماژول‌های استاندارد.

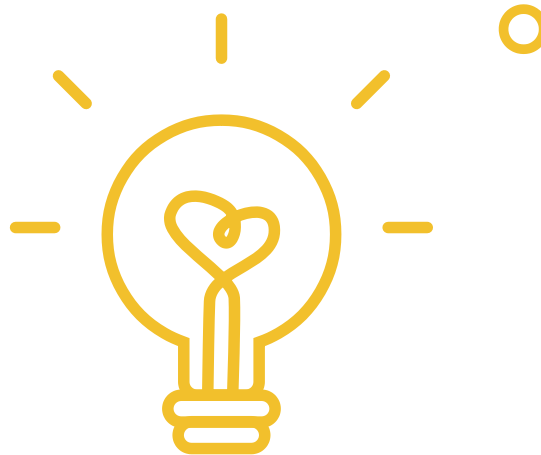
کد زیر مسیرهای جستجو را نمایش می‌دهد

```
import sys
```

```
print(sys.path)
```



بیشتر بدانید - namespace (فضای نام) در پایتون





بیشتر بدانید - namespace (فضای نام) در پایتون



namespace در پایتون یک سیستم نامگذاری است که برای جلوگیری از تداخل نام‌ها استفاده می‌شود.

هر نام در پایتون در یک فضای نام خاص وجود دارد که تعیین می‌کند آن نام به چه چیزی اشاره می‌کند.



۱. Built-in Namespace

```
# built-in namespace
```

```
print(len([1, 2, 3])) # len in built-in namespace
```

```
x = int("10")      # int in built-in namespace
```

```
print(dir()) # It shows you all variables name in the module
```



۲. Global Namespace

```
# global namespace
```

```
x = 10      # global variable
```

```
def my_func(): # global function
```

```
    pass
```

```
print(dir()) # It shows you all variables name in the module
```





بیشتر بدانید - namespace (فضای نام) در پایتون



۳. Local Namespace

```
# local namespace
def my_function():
    y = 20 # y is in local name space
    print(y)

print(dir()) # It shows you all variables name in the module
```



۴. Enclosing Namespace

```
# enclosing namespace
def outer_function():
    z = 30 # in enclosing namespace for inner_function

    def inner_function():
        print(z) # accessing to z in enclosing namespace

    inner_function()

outer_function()

print(dir()) # It shows you all variables name in the module
```





package





package



یک پکیج دایرکتوری ای است که شامل چندین ماژول و یک فایل `__init__.py` می باشد.



```
my_package/  
  __init__.py  
  module1.py  
  module2.py
```



فایل `__init__.py`:

نشان می دهد که دایرکتوری یک پکیج پایتون است

می تواند خالی باشد یا شامل کد مقداردهی اولیه

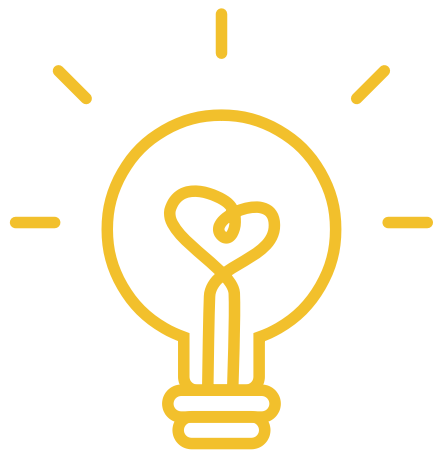
کنترل می کند چه چیزی هنگام `import` پکیج در دسترس باشد





تمرین ساخت package







pip




pip ابزار استاندارد مدیریت پکیج‌ها در پایتون است که برای نصب،



حذف و مدیریت پکیج‌های کتابخانه‌های پایتون استفاده می‌شود.



```
# ایجاد پروژه جدید
mkdir my_project
cd my_project
# ایجاد محیط مجازی
python3 -m venv venv
# فعالسازی محیط
source venv/bin/activate # Linux/Mac
# یا
venv\Scripts\activate    # Windows
# نصب پکیج‌ها
pip install requests django
# ذخیره لیست پکیج‌ها
pip freeze > requirements.txt
# وقتی پروژه را به سیستم دیگر منتقل کردید:
pip install -r requirements.txt
# pip list
# pip uninstall django
```





نوشتن یک داندلود منیجر ساده





بیشتر بدانید - poetry





بیشتر بدانید - poetry



Poetry یک ابزار مدرن برای مدیریت وابستگی‌ها و بسته‌ها در پایتون است

که جایگزین مناسبی برای pip به شمار می‌رود.



1. فایل `pyproject.toml`

جایگزین فایل‌های `requirements.txt`, `setup.py`, و `setup.cfg` می‌شود

و تمام اطلاعات پروژه را در خود نگهداری می‌کند.

2. قفل وابستگی‌ها (`poetry.lock`)

فایلی که نسخه دقیق هر وابستگی را قفل می‌کند تا محیط‌های مختلف سازگار بمانند.



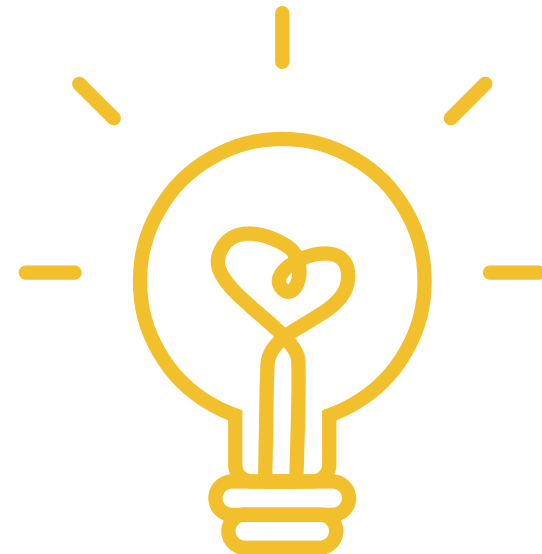


بیشتر بدانید - poetry



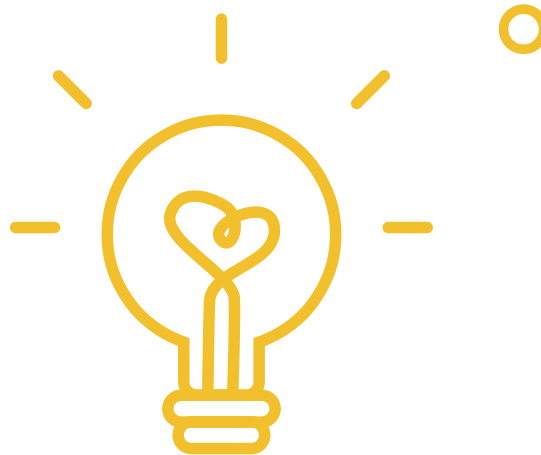
```
# ایجاد پروژه جدید
mkdir my_project
cd my_project
# ایجاد محیط مجازی
python3 -m venv venv
# فعالسازی محیط
source venv/bin/activate # Linux/Mac
# یا
venv\Scripts\activate # Windows
# نصب پکیج‌ها
poetry init
poetry add requests

# poetry remove requests
```





بیشتر بدانید - ماژول قابل نصب





THANK YOU

h.farhadi.py@gmail.com