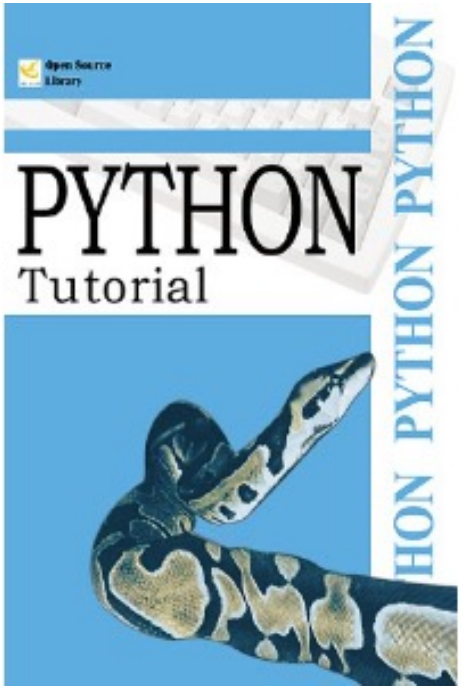
Functions



Dra. Mª Dolores Rodríguez Moreno

Objectives

Specific Objectives

- Understanding functions in Python

Source

- https://docs.python.org/3/reference/
- Charles R. Severance (www.pythonlearn.com)
- https://ellibrodepython.com/
- Python Tutorial - Tapa blanda. GuidoVan Rossum (2012)

Outline

- **Introduction**
- Conversions
- Function components
- Value vs Reference
- Multiple parameters
- Nested functions
- Annotations
- Lambda functions
- Decorators

Introduction

- There are two kinds of functions in Python
  - Built-in functions that are provided as part of Python - input(), type(), float(), max(), min(), int(), ...
  - Functions (user defined) that we define ourselves and then use them
- We treat the built-in function names like reserved words (i.e. we avoid them as variable names)
- Functions are **objects**, then you can store them in data structures

Introduction

- Since functions are objects, it means that they can be:
  - Assigned to variables
  - Passed as arguments to other functions
  - Returned from functions
- Functions in Python have attributes. Common built-in attributes :
  - __name__: The name of the function
  - __doc__: The function's docstring
  - __module__: The module in which the function is defined

Introduction (I)

- Functions in Python are objects and ultimately inherit from the base class 'object'

Functions are objects

```
def mi_funcion():
    pass

print(type(mi_funcion))  # <class 'function'>
print(isinstance(mi_funcion, object))  # True
```

We know already

- A function is some reusable code that takes arguments(s) as input, does some computation and then returns a result(s)
- We define a function using the ***def*** reserved word
  - We indent the body of the function
  - This defines the function but does not execute the body of the function
- We call/invoke the function by using the function name, parenthesis and arguments in an expression

Outline

- Introduction
- **Conversions**
- Function components
- Value vs Reference
- Multiple parameters
- Nested functions
- Annotations
- Lambda functions
- Decorators

Type Conversions

- When you put an integer and floating point in an expression the integer is implicitly converted to a float
- You can control this with the built in functions int() and float()

Type Conversions

Example:

```
print (float(99) / 100) # Output: 0.99

i = 42

type(i)  # Output: <type 'int'>

f = float(i) # print f   -- Output: 42.0

type(f)   # Output: <type 'float'>

a = 1 + 2 * float(3) / 4 - 5   # Output: -2.5
```

String Conversions

Example:

```
sval = '123'

type(sval)  #<type 'str'>

#print(sval + 1)   #Error

ival = int(sval)

type(ival)  # <type 'int'>

print (ival + 1)   # Output: 124

nsv = 'hello bob'

#niv = int(nsv)   #Error, no numeric characters
```

Outline

- Introduction
- Conversions
- **Function components**
- Value vs Reference
- Multiple parameters
- Nested functions
- Annotations
- Lambda functions
- Decorators

Arguments

- An argument is a value we pass into the function as its input when we call the function
- We use arguments so we can direct the function to do different kinds of work when we call it at different times
- We put the argument in parenthesis after the name of the function

Example:

```
Message = max ("my message")

#"my message" is the argument

#max is the name of the function
```

Parameters

- It is a variable which we use in the function definition
- It is a “handle” that allows the code in the function to access the arguments for a particular function invocation

Parameters

Example: Parameters

```
def greet(lang):

      if lang == 'es':

            print ('Hola')

      elif lang == 'fr':

            print ('Bonjour')

      else:

            print ('Hello')
```

Example: Arguments

```
greet('en')    #Hello

greet('es')    #Hola

greet('fr')    # Bonjour
```

Return Values

- It will often take its arguments, do some computation and return a value to be used as the value of the function call in the calling expression
- The ***return*** keyword is used for this

Example:

```
def greater(num):

      if num >= 0:

            return True

      else:

            return False
```

Example:

```
print (greater (10))    # True

print (greater (-10))   # False
```

Multiple Parameters/Arguments Values

- A function can take multiple parameters to perform operations

Example:

```
def add_numbers(a, b, c):

    return a + b + c

result = add_numbers(1, 2, 3)

print(result)  # Output: 6
```

Returning Multiple Values

- Python allows functions to return multiple values at once
- This feature enhances code readability and efficiency
- Method 1: using tuples

Multiple Return Values

- A function can return multiple values:

Example: Returning

```
def get_person_info():

    name = "John Doe"

    age = 30

    city = "New York"

    return name, age, city
```

Example: Calling

```
name, age, city = get_person_info()

print(name)   # Output: John Doe

print(age)    # Output: 30

print(city)   # Output: New York
```

Returning Multiple Values: Tuples

Example: Tuples

```
def get_coordinates():

    x = 10

    y = 20

    return x, y

coordinates = get_coordinates()

print(coordinates)  # Output: (10, 20)

print(f"x: {x}, y: {y}")  # Output: x: 10, y: 20
```

Returning Multiple Values: List

Example: List

```
def get_fibonacci(n):

    fib = [0, 1]

    for i in range(2, n):

        fib.append(fib[i-1] + fib[i-2])

    return fib

fibonacci_sequence = get_fibonacci(10)

print(f"First 10 Fibonacci numbers: {fibonacci_sequence}")
```

Returning Multiple Values: Dictionary

Example: Dictionary

```
def get_movie_info():

    return {

        "title": "La habitación de al lado",

        "director": ”Almodovar",

        "year": 2024,

        "rating": 9.2

    }

movie = get_movie_info()

print(f"{movie['title']} ({movie['year']}) directed by {movie['director']}")
```

Returning Multiple Values: Class

Example: Class

```
class Rectangle:

    def __init__(self, width, height):

        self.width = width

        self.height = height

def get_rectangle():

    return Rectangle(10, 5)

rect = get_rectangle()

print(f"Rectangle: width={rect.width}, height={rect.height}")
```

Void Functions

- A function that does not return any value
- It performs an action but does not return a result to the caller
- They do not use the return statement to return a value
- They perform an action such as printing, modifying global variables, or writing to a file
- If no return statement is used, they implicitly return None

Example:

```
def greet(lang):
        if lang == 'es':
                print ('Hola')
        elif lang == 'fr':
                print ('Bonjour')
        else:
                print ('Hello')
```

Exercise:

- Crea una función que calcule las raices de una ecuación de 2º grado: ax^2 + bx +c =0
- A partir de su coeficientes a,b y c
- Recuerde que si el discriminante es 0, la raiz es única x= -b/2a
- Devuelva dos valores, un único valor o un mensaje (raices imaginaries) dependiendo de los disintos casos que puedan ocurrir
- Utilice *isintance()* para saber lo que devuelve la función y dar el mensaje correspondiente

Outline

- Introduction
- Conversions
- Function components
- **Value vs Reference**
- Multiple parameters
- Nested functions
- Annotations
- Lambda functions
- Decorators

Value vs Reference

- Value: is the actual data stored in a variable
- Reference: is the address or location in memory where the data is stored
- In Python, **variables are references** to objects in memory
- This means when you assign a variable, you are assigning a reference to an object, not the actual value

Immutable vs. Mutable Types

- Immutable Types:
  - Definition: objects that cannot be changed after they are created
  - Examples: int, float, str, tuple
  - Behavior: when an immutable object is modified, a new object is created
- Mutable Types:
  - Definition: objects that can be changed after they are created
  - Examples: list, dict, set
  - Behavior: when a mutable object is modified, the original object is changed

Immutable Types (Pass by Value)

- When passed to a function, a copy of the reference is made
- Rebinding the reference inside the function does not affect the original object
- Result: The original variable remains unchanged

Example: immutable: int, float, str, tuple

```
x = 10

def funcion(entrada):

    entrada = 0

funcion(x)

print(x)   # Output: 10
```

Mutable Types (Pass by Reference)

- When passed to a function, a reference to the object is passed
- Modifying the object inside the function affects the original object
- Result: The original list x is modified

Example: mutable: list, dict, set

```
x = [10, 20, 30]

def function(entrada):

    entrada.append(40)

function(x)

print(x)  # Output: [10, 20, 30, 40]
```

Rebinding vs. Modifying Mutable Objects

- Rebinding “entr” to a new list does not change the original list x
- The reference “entr” now points to a new list, but x still points to the original list
- Result: The original list x remains unchanged

Example: Rebinding a mutable object

```
x = [10, 20, 30]

def function(entr):

    entr = []

function(x)

print(x)  # Output: [10, 20, 30]
```

Using id() to Understand References (I)

- The id() function returns a unique id for the object

Example: Different objects

```
x = 10

print(id(x))  # Example Output: 4349704528

def funcion(ent):

    ent = 0

    print(id(ent))  # Example Output: 4349704208

funcion(x)
```

Using id() to Understand References (II)

Example: The same object

```
x = [10, 20, 30]

print(id(x))  # Example Output: 4422423560

def funcion(entrada):

    entrada.append(40)

    print(id(entrada))  # Example Output: 4422423560

funcion(x)
```

Outline

- Introduction
- Conversions
- Function components
- Value vs Reference
- **Multiple parameters**
- Nested functions
- Annotations
- Lambda functions
- Decorators

Multiple Parameters with “*args”

- Use *args to pass a variable number of non-keyword arguments to a function
- non-keyword arguments = argumentos posicionales
- “*” means to pack them in a **tuple**
- Useful when you don´t know the number of arguments to the function
- The name “args” is a convection, you can change it

Multiple Parameters with *args

Example I: *args

```
def test_var_args(f_arg, *args):
    print("primer argumento normal:", f_arg)
    for arg in args:
        print("argumentos de *argv:", arg)
test_var_args('python', 'foo', 'bar')
```

Output

```
primer argumento normal: python
argumentos de *argv: foo
argumentos de *argv: bar
```

Multiple Parameters with *args (I)

Example II: *args

```
def print_numbers(*args):

    for number in args:

        print(number)

print_numbers(1, 2, 3, 4, 5) # Output: 1\n 2 \n 3\n 4\n 5\n
```

Multiple Parameters with **kwargs

- Use **kwargs to pass a variable number of keyword arguments to a function
- Key arguments = argumentos con nombre o de palabras clave
- ** means to pack the arguments in a dictionary
- The name “kwargs” is a convection, you can change it

Multiple Parameters with **kwargs

Example I: **kwargs

```
def print_info(**kwargs):

    for key, value in kwargs.items():

        print(f"{key}: {value}")

print_info(name="John", age=30, city="New York")

# Output: # name: John\n age: 30\n city: New York\n
```

Multiple Parameters with **kwargs (I)

Example II: **kwargs

```
def saludame(**kwargs):

    for key, value in kwargs.items():

        print("{0} = {1}".format(key, value))



saludame(nombre="Covadonga")

#Output: nombre = Covadonga
```

Multiple Parameters with **kwargs (II)

Example III: **kwargs

```
def crear_personaje(**kwargs):
    personaje = {

        "vida": 100,

        "fuerza": 50,

        "color": "azul"

    }

    personaje.update(kwargs)    #update the diccionary

    return personaje

jugador1 = crear_personaje(fuerza=80, color="rojo")
```

Multiple Parameters with **kwargs (III)

Example IV: **kwargs

```
def saludar(**kwargs):

    nombre = kwargs.get("nombre", "Invitado")

    edad = kwargs.get("edad", "desconocida")

    print(f"Hola {nombre}, tu edad es {edad}")

saludar(nombre="Ana")

saludar(edad=30)

saludar(nombre=”Pepe”, edad=36)
```

Combining *arg and **kwargs

- Let's consider this example:

Example I: Combining

```
def test_args_kwargs(arg1, arg2, arg3):

    print("arg1:", arg1)

    print("arg2:", arg2)

    print("arg3:", arg3)
```

Combining *args and **kwargs (I)

Example I: Combining

```
#With *args

args = ("dos", 3, 5)

test_args_kwargs(*args)

#output: arg1: dos \n arg2: 3 \n  arg3: 5  \n

# With  **kwargs:

kwargs = {"arg3": 3, "arg2": "dos", "arg1": 5}

test_args_kwargs(**kwargs)

#Output: arg1: 5 \n arg2: dos \n arg3: 3
```

Combining *args and **kwargs (II)

Example II: Combining

```
def print_details(*args, **kwargs):

    for arg in args:

        print(arg)

    for key, value in kwargs.items():

        print(f"{key}: {value}")

print_details(1, 2, 3, name="John", age=30, city="New York")
```

Combining *args and **kwargs (III)

Example III: Combining

```
def funcion_mixta(*args, **kwargs):

    print("args:", args)

    print("kwargs:", kwargs)

funcion_mixta(1, 2, 3, a=4, b=5)

# Salida:

# args: (1, 2, 3)

# kwargs: {'a': 4, 'b': 5}
```

Combining all parameters

- If you want to use all three types of input arguments to a function: normal, *args, and **kwargs, you must do so in the following order:

```
Function_name(fargs, *args, **kwargs)
```

Unpacking

Example: Unpacking

```
def saludar(nombre, edad):

    print(f"Hola, {nombre}. Tienes {edad} años.")

datos = ["Alice", 30]

info = {"nombre": "Bob", "edad": 25}

saludar(*datos)  # Desempaqueta la lista

saludar(**info)  # Desempaqueta el diccionario
```

Exercise:

- Crear una función calculadora que reciba un número variable de argumentos posicionales y de nombre
- La función debe realizar una operación sobre los números recibidos, basada en el argumento con nombre operacion, que puede ser "suma", "multiplicacion", "resta", o "division"

1. Define la función calculadora(*args, **kwargs)
2. Usa el valor del argumento con nombre (e.g. operacion="multiplicacion") para decidir qué operación realizar
3. Si no se especifica operacion, realiza la suma de los números

Outline

- Introduction
- Conversions
- Function components
- Value vs Reference
- Multiple parameters
- **Nested functions**
- Annotations
- Lambda functions
- Decorators

Nested functions

- You can make your functions nested
- Purpose:
  - Encapsulation: helps to hide the inner function from the outside world
  - Code organization: breaks down complex tasks into simpler, smaller functions

Nested functions

Example: Nested Function

```
def outer_function(text):

    def inner_function():

        print(text)

    inner_function()

outer_function("Hello, World!")

# Output: Hello, World!
```

Nested functions with Return Value

Example: Nested Function

```
def outer_function(x):

    def inner_function(y):

        return x + y

    return inner_function

adder = outer_function(10)

print(adder(5))  # Output: 15
```

Exercise:

- Crea una función: *crear_multiplicador (factor)* que reciba un número *factor* como argumento
- Defina una función interna llamada *multiplicar(numero)* que multiplique numero por factor
- Retorne la función interna *multiplicar()*
- Luego, usa la función *crear_multiplicador()* para:
  1. Crear una función llamada duplicar que duplique cualquier número que se le pase
  2. Crear una función llamada triplicar que triplique cualquier número que se le pase

Outline

- Introduction
- Conversions
- Function components
- Value vs Reference
- Multiple parameters
- Nested functions
- **Annotations**
- Lambda functions
- Decorators

Annotations

- A way of associating arbitrary metadata with function arguments and return values
- Purpose:
  - Provide additional information about the types and purposes of function arguments and return values
  - Improve code readability and support type hinting

Example: Annotations

```
def function_name(arg1: annotation1, arg2: annotation2) -> return_annotation:

    pass
```

Accessing Annotations

- Function annotations can be accessed using the *__annotations__* attribute
- The output is a dictionary where keys are parameter names and values are the annotations

Example: Annotations

```
def greet(name: str, age: int) -> str:

    return f"Hello, {name}. You are {age} years old."

print(greet.__annotations__)

# Output: {'name': <class 'str'>, 'age': <class 'int'>, 'return': <class 'str'>}
```

Combined with Default Values

Example: Annotations

```
def greet(name: str = "World") -> str:

    return f"Hello, {name}!”



def filtrar_pares(salida: list = []) -> list:

    return [i for i in salida if i % 2 == 0]



print(filtrar_pares([1, 2, 3, 4, 5, 6])) # Output: [2, 4, 6]

greet() # Output: Hello, World!
```

Customize Annotations

Example: Annotations – Custom Class

```
class ClassA:

    pass

def function(a: ClassA) -> ClassA:

    return a



a = ClassA()

function(a)
```

Forcing types?

- Python uses dynamic typing
- Annotations help specify expected types but do not enforce them

Example: Annotations – No error

```
# suma.py

def suma(a: int, b: int) -> int:

    return a + b

print(suma(7.0, 3))   #output 10
```

Static Checking

- Static Type Checking with *mypy*

Example: Annotations – Static type checking

```
$pip install mypy

$pip install typing
```

```
(base) MacBook-Pro-de-Maria-2:pythonProjects mariadr-moreno$ mypy suma.py
suma.py:5: error: Argument 1 to "suma" has incompatible type "float"; expected "int"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
(base) MacBook-Pro-de-Maria-2:pythonProjects mariadr-moreno$
```

Dynamic Checking

- Dynamic Type Checking with *isinstance*

Example: Annotations – Dynamic type checking

```
def add1(a: int, b: int) -> int:

    if not isinstance(a, int) or not isinstance(b, int):

        print(f"Arguments must be integers, got {type(a).__name__} and {type(b).__name__}")

    else:

        return a + b

print(add1(7.0, 3))
```

Outline

- Introduction
- Conversions
- Function components
- Value vs Reference
- Multiple parameters
- Nested functions
- Annotations
- **Lambda functions**
- Decorators

Lambda Functions

- Also known as anonymous functions
- Small, one-line functions without a name
- Defined using the 'lambda' keyword
- Can have any number of arguments, but only one expression

Why Use Lambda Functions?

- Concise way to create simple functions
- Useful for short, one-time use functions
- Can make code more readable in certain situations

- Can only contain expressions, not statements (e.g. cannot loops)
- Limited to a single expression
- Can be less readable for complex operations
- Debugging can be more difficult due to lack of a name

Lambda

Example: Lambda Function

```
# Regular function

def square(x):

    return x ** 2

# Equivalent lambda function

square_lambda = lambda x: x ** 2

print(square(4))         # Output: 16

print(square_lambda(4)) # Output: 16
```

Example: Lambda with Multiple Arguments

```
# Regular function

def multiply(x, y):

    return x * y



# Equivalent lambda function

multiply_lambda = lambda x, y: x * y



print(multiply(3, 4))        # Output: 12

print(multiply_lambda(3, 4)) # Output: 12
```

Example: Lambda with conditional expressions

```
# Lambda function with a conditional expression

max_lambda = lambda a, b: a if a > b else b



print(max_lambda(5, 3))  # Output: 5

print(max_lambda(2, 7))  # Output: 7
```

Example: functions as arguments – Annotations???

```
def aplicar_operacion(lista, operacion):

    return [operacion(x) for x in lista]

# Example: apply different operations

numeros = [1, 2, 3, 4, 5]

dobles = aplicar_operacion(numeros, lambda x: x * 2)

cuadrados = aplicar_operacion(numeros, lambda x: x ** 2)



print("Números originales:", numeros)

print("Dobles:", dobles)

print("Cuadrados:", cuadrados)
```

Example: functions as arguments

```
from typing import List, Callable, Any

def aplicar_operacion(lista: List, operacion: Callable) -> List:

      return [operacion(x) for x in lista]

numeros: List[int] = [1, 2, 3, 4, 5]

dobles: List[int] = aplicar_operacion(numeros, lambda x: x * 2)

cuadrados: List[int] = aplicar_operacion(numeros, lambda x: x ** 2)



print("Números originales:", numeros)

print("Dobles:", dobles)

print("Cuadrados:", cuadrados)
```

map function

- Returns a map object that we can convert to list or tuple

map

```
map(function, objeto iterable)
```

map function

Example

```
def cuadrado(numero):

 return numero * numero



lista = [1, 2, 3, 4, 5]

resultado = map(cuadrado, lista)



lista_resultado = list(resultado)

print(lista_resultado) #Output: [1, 4, 9, 16, 25]
```

Exercise

- Usa lambda y map() para convertir una lista de temperaturas de grados Celsius a Fahrenheit
  - *Fahrenheit = (grados centígrados × 9/5) +32*

Outline

- Introduction
- Conversions
- Function components
- Value vs Reference
- Multiple parameters
- Nested functions
- Annotations
- Lambda functions
- **Decorators**

Decorators

- Decorators are a powerful way to modify or extend the behavior of functions or classes without changing their source code
- They are functions that take another function (or class) as an argument and return a new, modified function (or class)
- Syntax: @decorator_name
- Lambda functions do not allow the use of decorator syntax

Mission

- Adding functionality to existing functions or methods
- Modifying the behaviour of functions or classes
- Logging function calls
- Measuring execution time
- Verifying permissions or authentication
- And many other uses...

Example: Decorator

```
def my_decorator(function):

    def wrapper():

        print("Before calling the function")

        function()

        print("After calling the function")

    return wrapper

@my_decorator

def greet():

    print("Hello!")

greet()
```

Example: Decorator with arguments

```
def what_i_do(times):

    def decorator(function):

        def wrapper(*args, **kwargs):

            for _ in range(times):

                result = function(*args, **kwargs)

            return result

        return wrapper

    return decorator

@what_i_do(3)

def greet(name):

    print(f"Hello, {name}")

greet("Alice")
```

Example I: Decorator for a tasks #import time

```
def medir_tiempo(func):

    def wrapper(*args, **kwargs):

        inicio = time.time()

        resultado = func(*args, **kwargs)

        fin = time.time()

        print(f"La función {func.__name__} tomó {fin - inicio:.4f} segundos.")

        return resultado

    return wrapper

@medir_tiempo

def calcular_suma(n):

     return sum(range(n))
```

Call the function

```
# Ejemplo de uso

print(calcular_suma(1000000))
```

Example: Class Decorators

```
def singleton(cls):

    instances = {}

    def get_instance(*args, **kwargs):

        if cls not in instances:

            instances[cls] = cls(*args, **kwargs)

        return instances[cls]

    return get_instance

@singleton

class Database:

    def __init__(self):

        print("Initializing the database")
```

Example: Class Decorators

```
db1 = Database()

# Prints: Initializing the database

db2 = Database()  # Prints nothing,
uses the same instance

print(db1 is db2)  # Prints: True
```

Example: Lambda and Decorator (WRONG)

```
def celsius_to_fahrenheit(func):

    return lambda c: func(c) * 9/5 + 32

# This will cause a syntax error

@celsius_to_fahrenheit

lambda c: c

# SyntaxError: invalid syntax
```

Example: Lambda and Decorator (RIGHT)

```
def celsius_to_fahrenheit(func):

    return lambda c: func(c) * 9/5 + 32

# Create a lambda function

celsius = lambda c: c

# Manually apply the decorator

fahrenheit = celsius_to_fahrenheit(celsius)

# Test the converter

print(fahrenheit(0))   # Output: 32.0

print(fahrenheit(100)) # Output: 212.0
```

Exercise…

- Create a decorator called log_event that will add a timestamp to the function call and print the function name along with its arguments. It will contain a nested function called “wrapper”
- Create lambda functions for different types of events (e.g., user_login, data_update, error_occurred)
- Use *args and **kwargs in your decorator (*wrapper* function) to handle any number of positional and keyword arguments
- Apply the decorator to the lambda functions and test them with various inputs
- Use *import time*

...Exercise

```
def log_event(func):
   pass
# Create lambda  functions for  events
user_login = lambda username: f"User {username} logged in"
#...
# Apply decorator to lambda functions
logged_user_login = log_event(user_login)
#....
# Try the functions
logged_user_login("Alice")
logged_data_update("user_count", 42)
logged_error_occurred(404, "Page not found")
```