sandeepsuryaprasad / **python_tutorials** Private

Code Issues Pull requests Actions Projects Security Ins

master python_tutorials / 5_Functions / **_functions.py** / Jump to Go to file

Sandeep Suryaprasad testing Latest commit 8d2eb85 on 4 Feb **History**

1 contributor

343 lines (273 sloc) | 10.4 KB Raw Blame

```python
import time

def greeting():
    print("Hello world")
    print('this is the body of the function')
    print('hello function!')

def greet():
    return "hello world"

def _greet(name):       # "name" is a variable or a parameter
    print(f"Hello {name}")

def greet_someone(name):
    return f"hello {name}"

def add(a, b):
    return a + b

# function with default values to the arguments
def _greet(name="world"):       # "name" is optional argument
    print(f"Hello {name}")

def greeting_(name, age, pay):
    # name, age and pay are called positional arguments
    print(f"Hello {name} you are {age} years of age and you get ${pay} as pay

def greeting_(name, age=26, pay=1000):
    # name, age and pay are called positional arguments
    print(f"Hello {name} you are {age} years of age and you get ${pay} as pay

def greet(name, debug=False):
    if debug:       # if debug == True
        print("You called greet function")
    print(f"hello {name}")

def greet(name, reverse=False, debug=False):
    if debug:
        print("You called greet function")
    if reverse:
        return f"hello {name[::-1]}"        # exits the function.
    return f"hello {name}"

def parse_string(line, delimiter=","):
    parts = line.split(delimiter)
    return parts

def greeting_(name, *, age, pay):
    # the parameters that are after * are to be called using keyword only
    # age and pay are KEYWORD ONLY Arguments, i.e. the value for age and pay
    print(f"Hello {name} you are {age} years of age and you get ${pay} as pay

def greet(name, *, reverse=False, debug=False):
    if debug:
        print("You called greet function")
    if reverse:
        return f"hello {name[::-1]}"        # exits the function.
    return f"hello {name}"

def greet(*, name, reverse=False, debug=False):
    if debug:
        print("You called greet function")
    if reverse:
        return f"hello {name[::-1]}"        # exits the function.
    return f"hello {name}"

def greet(name, /, *, reverse=False, debug=False):
    # the parameters that appears before "/" is positional only arguments
    if debug:
        print("You called greet function")
    if reverse:
        return f"hello {name[::-1]}"        # exits the function.
    return f"hello {name}"

# ---------------------------------------------------------------------------
# Variable number of positional (*args)
# * is used to grab arbitrary number of positional arguments!
def add(*args):
    total = 0.0
    # by convention we call variable number of positional arguments using par
    # * is used to collect excess arguments
    for item in args:
        total = total + item
    return total

print(add())
print(add(1))
print(add(1, 2))
print(add(10, 30, 45))
print(add(1000, 46273, 84545, 9834958, 4587583))
nums = [1, 2, 3, 4]
print(add(*nums))
# ---------------------------------------------------------------------------
def greet(*names):
    for name in names:
        print(f'hello {name}')

greet("steve")      # one argument
greet("steve", "bill")  # two arguments
greet("steve", "bill", "gates", "jobs", "joe")      # five arguments
greet() # zero arguments
# ---------------------------------------------------------------------------
# Variable number of keyword arguments (**kwargs)
# * is used to grab arbitrary number of positional arguments!
def greet(name, **info):
    print(f'hello {name} below is your information')
    for key, value in info.items():
        print(f'{key}: {value}')

greet("Steve", phone=1234567890, city="Bangalore", country="India")     # Thr
greet("Steve", state="Karnataka")       # One arbitrary keyword argument
greet("Steve")      # Zero keyword argument
# ---------------------------------------------------------------------------

def func(a, *args):
    print(a, args)

```

```python
# Keyword variable arguments (**kwargs)
def func2(a, **kwargs):
    print(a, kwargs)

# Combining both
def anyargs(*args, **kwargs):
    print(args)     # Tuple
    print(kwargs)   # Dictionary

anyargs(1, 2, 3, fname='steve', lname='jobs')

# Unpacking arguments
def greet(name, age, pay):
    print(f'Hello {name} you are {age} years and you have {pay} pay')

data = ['Steve', 26, 1000]

greet(data[0], data[1], data[2])
greet(*data)    # Equivelent to greet("Steve", 26, 1000)

d_data = {"name": "steve", "age": 26, "pay": 1000}
greet(d_data['name'], d_data['age'], d_data['pay'])
greet(**d_data)     # Equivelent to greet(name="Steve", age=26, pay=1000)

# Returning Multiple Values from a Function
def div(a, b):
        r = a % b
        q = a / b
        return r, q  # returns a tuple

remainder, quotent = div(4, 2)

# passing reference of one function to another function
def greet():
    return "Hello world"

def spam(func):
    return func()

a = spam(greet)
# 1. Ttionhe reference of "greet" function is passed to "spam" func.
# 2. "spam" function is invoking or executing "greet" function
# 3. "spam" function is also know as "callback" function. Meaning, the functi
# function "greet" which is passed to it.

# "spam" retuns the reference of the function that is being passed to it.
def spam(func):
    return func

b = spam(greet)
b() # invoking "greet" function
# both "b" and "greet" are pointing to same function object in the memory

# Passing function to another function. Functions as "First class" objects.
def _delay(_func, _time, *args, **kwargs):
    time.sleep(_time)
    print(args)
    print(kwargs)
    result = _func(*args, **kwargs)
    return result
# ---------------------------------------------------------------------------
# Function Annotations
# Annotations are only type hints. But it does not enforce type check!
def add(a: int, b: int) -> int:
    return a + b

def greetings(name: str, age: int, pay: float, isMarried: bool) -> None:
    print(f"Hello {name} You are {age} years old and your is {pay}")
    if isMarried:
        print('Congratulations')
    else:
        print('You are free')

def greet(name: str = "Spider") -> None:
    print(f'Hello {name}')

# ---------------------------------------------------------------------------
# Default values are evaluated only once at the time of function defnition
age = 10
def myinfo(my_name, my_age=age):
    print(my_name, my_age)

print(myinfo('steve', my_age=50))      # Prints (steve, 50)
print(myinfo('steve'))      # Prints(steve, 10)
age = 20
print(myinfo('steve'))      # Prints (steve, 10)

# Default arguments are evaluated only ONCE
"""
    names=[ ] in the function declaration makes Python essentially do this:
```

```python
    1. This function has a parameter named "names" its default argument is [
        let's set this particular [ ] aside and use it anytime there's no par
    2. Every time the function is called, create a variable "names", and assi
        the passed parameter or the value we set aside earlier
"""
def func(names=[ ]):  # making mutable data as default value
    names.append(1)
    return names

func()  # returns [1]
func()  # returns [1, 1]
func()  # returns [1, 1, 1]
func([10, 20, 30, 40]) # returns [10, 20, 30, 40, 1]

# Correct version
def func(names = None):
    if names is None:
        names = [ ]
    names.append(1)
    return names

func()  # returns [1]
func()  # returns [1]
func()  # returns [1]
func([10, 20, 30, 40]) # returns [10, 20, 30, 40]
# -------------------------------------------------------------------------------

# lambda expressions/functions
# General Syntax
# lambda args: expression         # (expression is something which evaluates

def add(a, b):
    return a+b  # Single expression function

def func(a, b):
    return a ** 2 + b ** 2 + 2 * a * b

def func2(a, b, c):
    return 2*a + 3*b + 4*c

def last(item):
    return item[-1]

# lambda expressions or ananoymous functions
# lambda args_list: expression
```

```python
add = lambda a, b: a + b
func = lambda a, b: a ** 2 + b ** 2 + 2 * a * b
func2 = lambda a, b, c: 2*a + 3*b + 4*c
last = lambda item: item[-1]
# ---------------------------------------------------------------------------
# Passing Immutable data to functions
a = 10
def spam(some_number):
    some_number = some_number + 1
    print(some_number)

spam(a)  # Prints 11
print(a)  # Prints 10
# ---------------------------------------------------------------------------
# Passing Mutable data to functions
a = [10]

def spam(some_list):
    some_list.append(20)
    print(some_list)

spam(a)  # Prints [10, 20]
print(a)  # Prints [10, 20]

# ---------------------------------------------------------------------------
numbers = [5, 1, 3, 2, 0, 7, 6]

def smallest(items, n):
    items.sort()
    return items[:n]

"""
1. When an Immutable object is passed to a function, python acts as
call by value.
2. When a Mutable object is passed to a function, python acts as call
by reference.
3. Python is neither call by value nor call by reference. It all depends
on the type of the object that is being passed to the function
"""

# ---------------------------------------------------------------------------
a = 10  # Global variable

# defining the function
def func(b):
    return a + b

a = 20      # Re-assigning new value to the global variable

func(10) # prints 30    # executing the function
# this is called as late-binding
# The "func" uses the value of "a" that happens to be at the time of evaluati

# ---------------------------------------------------------------------------
_a = 200
# If it is important to use the value of the variable at the time of function
# use default arguments.
def func2(b, a=_a):
    return a + b

_a = 100      # Re-assigning new value to the global variable

func2(10)       # prints 210
# In the function "func2" the parameter "a" takes the value that is assigned 
# ---------------------------------------------------------------------------

# You can attach arbitrary attributes to the function after the function is d

def add(a, b):
    add.count += 1
    return a+b

def sub(a, b):
    sub.count += 1
    return a-b

# Attach the attributes to the function
add.count = 0
sub.count = 0

add(1, 2)
add(10, 20)

print(add.count)    # prints count = 2

sub(1, 2)
sub(1, 3)
sub(1, 4)
sub(1, 5)

```

```python
print(sub.count)    # prints count = 4
```