

[sandeepsuryaprasad](#) / [python_tutorials](#) Private[Code](#) [Issues](#) [Pull requests](#) [Actions](#) [Projects](#) [Security](#) [Ins](#)

master ▾

[python_tutorials](#) / [5_Functions](#) / [_functions.py](#)[Go to file](#)

...

[/ <> Jump to ▾](#)**Sandeep Suryaprasad** testing

Latest commit 8d2eb85 on 4 Feb

[History](#)[1](#) contributor

343 lines (273 sloc) | 10.4 KB

[Raw](#)[Blame](#)

```
1  import time
2
3  def greeting():
4      print("Hello world")
5      print('this is the body of the function')
6      print('hello function!')
7
8  def greet():
9      return "hello world"
10
11 def _greet(name):          # "name" is a variable or a parameter
12     print(f"Hello {name}")
13
14 def greet_someone(name):
15     return f"hello {name}"
16
17 def add(a, b):
18     return a + b
19
20 # function with default values to the arguments
21 def _greet(name="world"):    # "name" is optional argument
22     print(f"Hello {name}")
23
24 def greeting_(name, age, pay):
25     # name, age and pay are called positional arguments
26     print(f"Hello {name} you are {age} years of age and you get ${pay} as pay")
27
```

```
28 def greeting_(name, age=26, pay=1000):
29     # name, age and pay are called positional arguments
30     print(f"Hello {name} you are {age} years of age and you get ${pay} as pay
31
32 def greet(name, debug=False):
33     if debug:          # if debug == True
34         print("You called greet function")
35     print(f"hello {name}")
36
37 def greet(name, reverse=False, debug=False):
38     if debug:
39         print("You called greet function")
40     if reverse:
41         return f"hello {name[::-1]}"          # exits the function.
42     return f"hello {name}"
43
44 def parse_string(line, delimiter=","):
45     parts = line.split(delimiter)
46     return parts
47
48 def greeting_(name, *, age, pay):
49     # the parameters that are after * are to be called using keyword only
50     # age and pay are KEYWORD ONLY Arguments, i.e. the value for age and pay
51     print(f"Hello {name} you are {age} years of age and you get ${pay} as pay
52
53 def greet(name, *, reverse=False, debug=False):
54     if debug:
55         print("You called greet function")
56     if reverse:
57         return f"hello {name[::-1]}"          # exits the function.
58     return f"hello {name}"
59
60 def greet(*, name, reverse=False, debug=False):
61     if debug:
62         print("You called greet function")
63     if reverse:
64         return f"hello {name[::-1]}"          # exits the function.
65     return f"hello {name}"
66
67 def greet(name, /, *, reverse=False, debug=False):
68     # the parameters that appears before "/" is positional only arguments
69     if debug:
70         print("You called greet function")
71     if reverse:
72         return f"hello {name[::-1]}"          # exits the function.
```

```
73     return f"hello {name}"
74
75 # -----
76 # Variable number of positional (*args)
77 # * is used to grab arbitrary number of positional arguments!
78 def add(*args):
79     total = 0.0
80     # by convention we call variable number of positional arguments using par
81     # * is used to collect excess arguments
82     for item in args:
83         total = total + item
84     return total
85
86 print(add())
87 print(add(1))
88 print(add(1, 2))
89 print(add(10, 30, 45))
90 print(add(1000, 46273, 84545, 9834958, 4587583))
91 nums = [1, 2, 3, 4]
92 print(add(*nums))
93 # -----
94 def greet(*names):
95     for name in names:
96         print(f'hello {name}')
97
98 greet("steve")          # one argument
99 greet("steve", "bill")  # two arguments
100 greet("steve", "bill", "gates", "jobs", "joe")    # five arguments
101 greet() # zero arguments
102 # -----
103 # Variable number of keyword arguments (**kwargs)
104 # * is used to grab arbitrary number of positional arguments!
105 def greet(name, **info):
106     print(f'hello {name} below is your information')
107     for key, value in info.items():
108         print(f'{key}: {value}')
109
110 greet("Steve", phone=1234567890, city="Bangalore", country="India")    # Thr
111 greet("Steve", state="Karnataka")    # One arbitrary keyword argument
112 greet("Steve")    # Zero keyword argument
113 # -----
114
115 def func(a, *args):
116     print(a, args)
117
```

```
118 # Keyword variable arguments (**kwargs)
119 def func2(a, **kwargs):
120     print(a, kwargs)
121
122 # Combining both
123 def anyargs(*args, **kwargs):
124     print(args)      # Tuple
125     print(kwargs)    # Dictionary
126
127 anyargs(1, 2, 3, fname='steve', lname='jobs')
128
129 # Unpacking arguments
130 def greet(name, age, pay):
131     print(f'Hello {name} you are {age} years and you have {pay} pay')
132
133 data = ['Steve', 26, 1000]
134
135 greet(data[0], data[1], data[2])
136 greet(*data)      # Equivalent to greet("Steve", 26, 1000)
137
138 d_data = {"name": "steve", "age": 26, "pay": 1000}
139 greet(d_data['name'], d_data['age'], d_data['pay'])
140 greet(**d_data)    # Equivalent to greet(name="Steve", age=26, pay=1000)
141
142 # Returning Multiple Values from a Function
143 def div(a, b):
144     r = a % b
145     q = a / b
146     return r, q    # returns a tuple
147
148 remainder, quotient = div(4, 2)
149
150 # passing reference of one function to another function
151 def greet():
152     return "Hello world"
153
154 def spam(func):
155     return func()
156
157 a = spam(greet)
158 # 1. The reference of "greet" function is passed to "spam" func.
159 # 2. "spam" function is invoking or executing "greet" function
160 # 3. "spam" function is also known as "callback" function. Meaning, the function
161 #    "greet" which is passed to it.
162
```

```
163 # "spam" returns the reference of the function that is being passed to it.
164 def spam(func):
165     return func
166
167 b = spam(greet)
168 b() # invoking "greet" function
169 # both "b" and "greet" are pointing to same function object in the memory
170
171 # Passing function to another function. Functions as "First class" objects.
172 def _delay(_func, _time, *args, **kwargs):
173     time.sleep(_time)
174     print(args)
175     print(kwargs)
176     result = _func(*args, **kwargs)
177     return result
178 # -----
179 # Function Annotations
180 # Annotations are only type hints. But it does not enforce type check!
181 def add(a: int, b: int) -> int:
182     return a + b
183
184 def greetings(name: str, age: int, pay: float, isMarried: bool) -> None:
185     print(f"Hello {name} You are {age} years old and your is {pay}")
186     if isMarried:
187         print('Congratulations')
188     else:
189         print('You are free')
190
191 def greet(name: str = "Spider") -> None:
192     print(f'Hello {name}')
193
194 # -----
195 # Default values are evaluated only once at the time of function definition
196 age = 10
197 def myinfo(my_name, my_age=age):
198     print(my_name, my_age)
199
200 print(myinfo('steve', my_age=50))      # Prints (steve, 50)
201 print(myinfo('steve'))                 # Prints(steve, 10)
202 age = 20
203 print(myinfo('steve'))                 # Prints (steve, 10)
204
205 # Default arguments are evaluated only ONCE
206 """
207     names=[ ] in the function declaration makes Python essentially do this:
```

```
208     1. This function has a parameter named "names" its default argument is [
209         let's set this particular [ ] aside and use it anytime there's no par
210     2. Every time the function is called, create a variable "names", and assi
211         the passed parameter or the value we set aside earlier
212     """
213     def func(names=[ ]): # making mutable data as default value
214         names.append(1)
215         return names
216
217     func() # returns [1]
218     func() # returns [1, 1]
219     func() # returns [1, 1, 1]
220     func([10, 20, 30, 40]) # returns [10, 20, 30, 40, 1]
221
222     # Correct version
223     def func(names = None):
224         if names is None:
225             names = [ ]
226             names.append(1)
227         return names
228
229     func() # returns [1]
230     func() # returns [1]
231     func() # returns [1]
232     func([10, 20, 30, 40]) # returns [10, 20, 30, 40]
233     # -----
234
235     # lambda expressions/functions
236     # General Syntax
237     # lambda args: expression          # (expression is something which evaluates
238
239     def add(a, b):
240         return a+b # Single expression function
241
242     def func(a, b):
243         return a ** 2 + b ** 2 + 2 * a * b
244
245     def func2(a, b, c):
246         return 2*a + 3*b + 4*c
247
248     def last(item):
249         return item[-1]
250
251     # lambda expressions or anonymous functions
252     # lambda args_list: expression
```

```
253 add = lambda a, b: a + b
254 func = lambda a, b: a ** 2 + b ** 2 + 2 * a * b
255 func2 = lambda a, b, c: 2*a + 3*b + 4*c
256 last = lambda item: item[-1]
257 # -----
258 # Passing Immutable data to functions
259 a = 10
260 def spam(some_number):
261     some_number = some_number + 1
262     print(some_number)
263
264 spam(a) # Prints 11
265 print(a) # Prints 10
266 # -----
267 # Passing Mutable data to functions
268 a = [10]
269
270 def spam(some_list):
271     some_list.append(20)
272     print(some_list)
273
274 spam(a) # Prints [10, 20]
275 print(a) # Prints [10, 20]
276
277 # -----
278 numbers = [5, 1, 3, 2, 0, 7, 6]
279
280 def smallest(items, n):
281     items.sort()
282     return items[:n]
283
284 """
285 1. When an Immutable object is passed to a function, python acts as
286 call by value.
287 2. When a Mutable object is passed to a function, python acts as call
288 by reference.
289 3. Python is neither call by value nor call by reference. It all depends
290 on the type of the object that is being passed to the function
291 """
292
293 # -----
294 a = 10 # Global variable
295
296 # defining the function
297 def func(b):
```

```
298     return a + b
299
300 a = 20      # Re-assigning new value to the global variable
301
302 func(10) # prints 30      # executing the function
303 # this is called as late-binding
304 # The "func" uses the value of "a" that happens to be at the time of evaluation
305
306 # -----
307 _a = 200
308 # If it is important to use the value of the variable at the time of function
309 # use default arguments.
310 def func2(b, a=_a):
311     return a + b
312
313 _a = 100      # Re-assigning new value to the global variable
314
315 func2(10)      # prints 210
316 # In the function "func2" the parameter "a" takes the value that is assigned
317 # -----
318
319 # You can attach arbitrary attributes to the function after the function is defined
320
321 def add(a, b):
322     add.count += 1
323     return a+b
324
325 def sub(a, b):
326     sub.count += 1
327     return a-b
328
329 # Attach the attributes to the function
330 add.count = 0
331 sub.count = 0
332
333 add(1, 2)
334 add(10, 20)
335
336 print(add.count)      # prints count = 2
337
338 sub(1, 2)
339 sub(1, 3)
340 sub(1, 4)
341 sub(1, 5)
342
```



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