11.1_Functions(2)

March 27, 2023

1 Introduction to Python for Open Source Geocomputation

- Instructor: Dr. Wei Kang
- Class Location and Time: ENV 336, Mon & Wed 12:30 pm - 1:50 pm

Content:

- Arguments in Functions
  - positional
  - keyword
  - variable length arguments

2 Functions

Functions are ways we can extend Python by writing code to add functionality that we would like to **reuse**.

- Required:
  - function keyword `def`
  - function name
  - solution: statements/expressions
- Optional
  - Argument(s) (input)/parameters
    - Arguments: values passed to the function
    - Parameters: variables in the function that were assigned to by arguments
  - Return values (output)

2.1 Function Argument Types

- positional (what we have defined so far)
- keyword
- variable length positional
- variable length keyword

2.1.1 Positional Arguments

```
def power_positional(x, exponent):
    return x**exponent
```

- The position of each positional argument is critical
  - `power_positional(2,3)` is different from `power_positional(3,2)`
  - the first argument passed in the funciton is assigned to the local variable `x`
  - the second argument passed in the funciton is assigned to the local variable `exponent`
- Positional arguments are **required**
  - `power_positional(3)` will throw an error

[7]:
```
def power_positional(x, exponent):
    return x**exponent
```

[8]:
```
power_positional(x=2, exponent=3)
```

[8]: 8

[2]:
```
power_positional(2, 3)
```

[2]: 8

[3]:
```
power_positional(3, 2)
```

[3]: 9

[4]:
```
power_positional(3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[4], line 1
----> 1 power_positional(3)

TypeError: power_positional() missing 1 required positional argument: 'exponent
```

2.1.2 Keyword Arguments

```
def power_keyword(x=2, exponent=3):
    return x**exponent
```

Keyword parameters can serve two uses:

- Define default values for parameters
- When we call functions in this way, the order (position) of the arguments can be changed

[5]:
```
def power_keyword(x=2, exponent=3):
    return x**exponent
```

[6]:
```
power_keyword() #default values 2,3 are used
```

[6]: 8

[9]:
```
power_keyword(x=2, exponent=3)
```

[9]: 8

[10]:
```
power_keyword(x=2)
```

[10]: 8

We may pass the value to a keyword argument and leave the other one with default value

[11]:
```
power_keyword(exponent=3)
```

[11]: 8

[ ]:
```
def power_keyword(x=2, exponent=3):
    return x**exponent
```

[12]:
```
power_keyword(exponent=3, x=2)
```

[12]: 8

We can change the order of the keyword arguments

[13]:
```
power_keyword(2, 3)
```

[13]: 8

We can also use the keywords implicitly and their positions explicitly

[ ]:
```
def power_keyword(x=2, exponent=3):
    return x**exponent
```

[14]:
```
power_keyword(2, exponent=3)
```

[14]: 8

[15]:
```
power_keyword(x=2, 3)
```

```
  Cell In[15], line 1
    power_keyword(x=2, 3)
                       ^
SyntaxError: positional argument follows keyword argument
```

positional arguments have to come before the keyword arguments

[16]:
```
def power_keyword(x=2, exponent=3):
    return x**exponent
```

[17]:
```
power_keyword(2, x=3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[17], line 1
----> 1 power_keyword(2, x=3)

TypeError: power_keyword() got multiple values for argument 'x'
```

[18]:
```
power_keyword(2, exponent=3)
```

[18]:
```
8
```

2.1.3 Combining Positional and Keyword Arguments in Defining a function

We can combine these two types of arguments in a defining a function

- using positional arguments to specify **required** parameters
- using keyword arguments to define **optional** parameters
- positional arguments have to **precede** the keyword arguments.

```
def power_combined(x, exponent=3):
    return x**exponent
```

[19]:
```
def power_combined(x, exponent=3):
    return x**exponent
```

[20]:
```
power_combined(2)
```

[20]:
```
8
```

[21]:
```
power_combined(2, 3)
```

[21]:
```
8
```

[ ]:
```
def power_combined(x, exponent=3):
    return x**exponent
```

[22]:
```
power_combined(7, 5)
```

[22]: 16807

[23]:
```
power_combined(7, exponent=5)
```

[23]: 16807

2.1.4 Group Exercise:

Suppose the cover price of a book is $24.95, but bookstores get a 40% discount. Shipping costs \$3 for the first copy and 75 cents for each additional copy. What is the total wholesale cost for 60 copies? Write a function `total_cost` with five arguments/parameters:

- `price`: cover price of a book
- `discount`: discount for each book
- `shipping_first`: shipping cost for the first copy
- `shipping_additional`: shipping cost for each additional copy
- `copies`: number of copies to purchase

`copies` and `price` are keyword arguments while the other three are positional arguments.

> Once you complete the function, caculate the total cost for cover price (\$24.95), discount(10%), shipping for first copy (\$5) and each additional copy (\$0.5), and copies (60)
>
> raise your hand when you are done!

[ ]:
```
def calc_total_cost(discount, shipping_first, shipping_additional, copies=60,␣
↪price = 24.95):
    total = price * (1-discount) * copies + (shipping_first+ (copies-1) *␣
↪shipping_additional)
    return total
```

[ ]:
```
calc_total_cost(0.1, 5, 0.5)
```

[ ]:
```
calc_total_cost(0.1, 5, 0.5, 60, 24.95)
```

2.1.5 Variable Length Positional Arguments

```
def power_vapa(x,*names,exponent=2):
    print('x: ',x)
    print('exponent: ',exponent)
    for name in names:
        print(name)
```

- used to accept an undetermined (at definition time) number of positional arguments

- they are tucked into a **tuple**
- they have to come after positional arguments
- they can come after keyword arguments

[ ]:
```
def power_vapa(x,*names,exponent=2):
    print('x: ',x)
    print('exponent: ',exponent)
    for name in names:
        print(name)
```

[ ]:
```
power_vapa(2,3,6,7,7)
```

[ ]:
```
power_vapa(2,3,6,7,7,10,2,3,1,243,254)
```

keywords arguments can be omitted when calling the function

[ ]:
```
power_vapa(2,3,6,7,7,exponent=100)
```

keywords arguments before Variable Length Positional Arguments?

[ ]:
```
def power_vapa_r(x,exponent=2,*names):
    print('x: ',x)
    print('exponent: ',exponent)
    for name in names:
        print(name)
```

[ ]:
```
power_vapa_r(2,3,6,7,7)
```

if keywords arguments comes before Variable Length Positional Arguments, they are treated as positional arguments and are required when calling the function

[ ]:
```
power_vapa_r(2,3,"python", 6,7,7)
```

`print()` is a function that accepts Variable Length Positional Arguments

[ ]:
```
print(1,2,3,4,"happy")
```

[ ]:
```
print(1,23,4,5,2,3,52,3,"astring", [12])
```

2.1.6 Variable Length Keyword Arguments

```
def power_vaka(x,exponent=2,**theRest):
    print('x: ',x)
    print('exponent: ',exponent)
    for key,value in theRest.items():
        print(key,value)
```

- used to accept an undetermined (at definition time) number of keyword arguments
- they are tucked into a **dictionary**

- they have to come after positional, Variable Length Positional Arguments, and keywords arguments

[ ]:
```
def power_vaka(x,exponent=2,**theRest):
    print('x: ',x)
    print('exponent: ',exponent)
    for key,value in theRest.items():
        print(key,value)
```

[ ]:
```
theRest = dict([("university", "UNT"),("state","TX")])
theRest
```

[ ]:
```
power_vaka(3,22, university="UNT", state="TX")
```

[ ]:
```
power_vaka(3,exponent=22, university="UNT", state="TX")
```

Mixing Variable Length Keyword and positional Arguments in one function

[ ]:
```
def power_vapka(x,exponent=2, *names, **theRest):
    print('x: ',x)
    print('exponent: ',exponent)
    for a in names:
        print(a)
    for key,value in theRest.items():
        print(key,value)
```

[ ]:
```
power_vapka(3,2, "python", 1,23, university="UNT", state="TX")
```

2.2 global and local variables

- Inside a function, variables and parameters are local
  - it only exists inside the function.
  - When the function terminates, the variables/paramters inside the function is destroyed. If we try to print it, we get an error.

[ ]:
```
def times2(number):
    new_number = number * 2
    return new_number
```

[ ]:
```
print(number)
```

[ ]:
```
print(new_number)
```

The variables defined in the function exist only in its *namespace*.

Here, in the *global namespace* we get a `NameError` when trying to access the variables `number` or `new_number` because they have only been defined within the `times2()` function.

How about calling the function first?

[ ]:
```
times2(number=5)
```

[ ]:
```
print(number)
```

[ ]:
```
print(new_number)
```

As you can see `number` is still not defined in the global namespace.

Why does Python work this way?

Well, as it turns out, the benefit of having a separate namespace for functions is that we can define a variable in the global namespace, such as `number` and not need to worry about its name within a function, or the use of a function changing its value.

Inside the function, the value that is passed will be known as `number`, but modifying that value will not alter a variable of the same name in the global namespace.

Let's have a look at another example using a modified `times2()` function we can call `times2v2()`.

[ ]:
```
def times2v2(number):
    number = number * 2
    return number
```

Let's now define a variable `number` in the global namespace and use our function to multiply it by 2.

[ ]:
```
number = 15
```

[ ]:
```
times2v2(number)
```

[ ]:
```
number
```

As you can see, the value of the variable `number` in the global namespace was set to 15 and remains 15 after using the `times2v2()` function.

Although there is a variable inside that function with the same name as the value in the global namespace, using the function assigns the value of `number` inside the function and manipulates that value only inside the function.

```
Be aware that it is possible to access variable values in functions that have been defined in t
This is because Python will search for variables defined with a given name first inside the fu
If such a value is found, it can be used by the function, which could be dangerous.
```

Let's look at an example of behavior in a function that may be unexpected.

[ ]:
```
def times2v2plus(number):
    number = number * 2 + value
    return number
```

[ ]:
```
times2v2plus(1)
```

[ ]:
```
value = 10
```

[ ]:
```
times2v2plus(1)
```

Although `value` was not passed to `times2v2plus()` it is defined in the global namespace and thus can be used by our example function.

Be careful!

[ ]:
```
def times2v2plus(number):
    value = value *2
    number = number * 2 + value
    return number
```

[ ]:
```
times2v2plus(1)
```

[ ]:
```
value
```

We run into a UnboundLocalError because when we make an assignment to a variable in a scope (`value = value *2`), that variable becomes local to that scope and shadows any similarly named variable in the outer scope. Since(`value = value *2`) assigns a new value to `value`, the compiler recognizes it as a local variable. https://docs.python.org/3/faq/programming.html#why-am-i-getting-an-unboundlocalerror-when-the-variable-has-a-value

2.2.1 Further reading on namespaces and variables scopes

For those who are interested, more information about namespaces and variables scopes can be found on the Real Python website.

2.3 Functions within a function

An example: Calculate $a^2 + b^2$

[ ]:
```
def square(a):
    return a**2
```

[ ]:
```
def calc(a, b):
    return square(a) + square(b)
```

[ ]:
```
calc(10, 200)
```

2.4 Function docstrings

A docstring is a string that starts on the first new line immediately after the declaration of a function or a class. Like the body of the function or class, the docstring must be indented 4 spaces. Typically, a docstring is contained within a block string, set off by tripe quotes " " "

[ ]:
```
def function(x, y):
    """
```

```
    A one-line summary that does not use variable names or the function name.

    Parameters
    ----------
    x : type
        Description of parameter `x`.
    y
        Description of parameter `y` (with type not specified).


    Returns
    -------
    err_code : int
        Non-zero value indicates error code, or zero on success.
    err_msg : str or None
        Human readable error message, or None on success.
    """
    pass
```

[ ]:
```
function(1,2)
```

[ ]:
```
def determine_multiple(a):
    """Determine whether a given integer is a multiple of 2 or/and 3.

    Parameters
    ----------
    a : int
        An integer.

    Return
    ------
      : str
        Descriptions.

    """
    ### BEGIN SOLUTION
    if a % 6 == 0:
        return "The number is a multiple of 6"
    elif a % 2 == 0:
        return "The number is a multiple of 2"
    elif a % 3 == 0:
        return "The number is a multiple of 3"
    else:
        return "The number is not a multiple of 2 or 3"
    ### END SOLUTION
```

[ ]:
```

```