COMP 150: Functions

Week 12, Lecture 18/Lab 08

Agenda

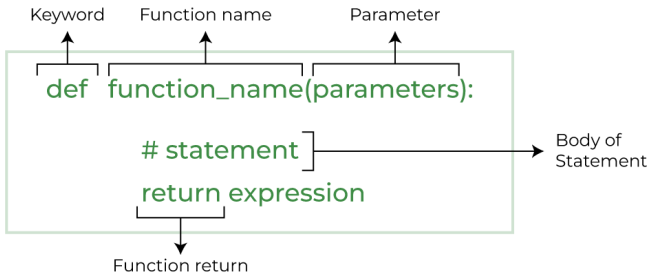
1. Exam next week
2. Recursive Functions
3. Lab 08

Exam Next Week

- April 10 (Friday)
- 50 minutes
- Two written questions (10-15 minutes), closed world
  - No deep details, but any discussion is an option
- Five practical problems (25-30 minutes), open internet, *no AI*
  - Variables
  - Loops/conditionals
  - Data read-in (CSVs)
  - Pandas, Plotly
  - Functions

Terms for Functions

- *Define* it using `def`
- *Return from it* when we're finished in the function
- *Call* it using its name and parentheses
- *Parameters* are what we call the things given to a function
- *Arguments* are the values we give as parameters
- We *pass* arguments to a function



The Function

- Reusable unit of data = variable
- Reusable unit of code = function

Why?

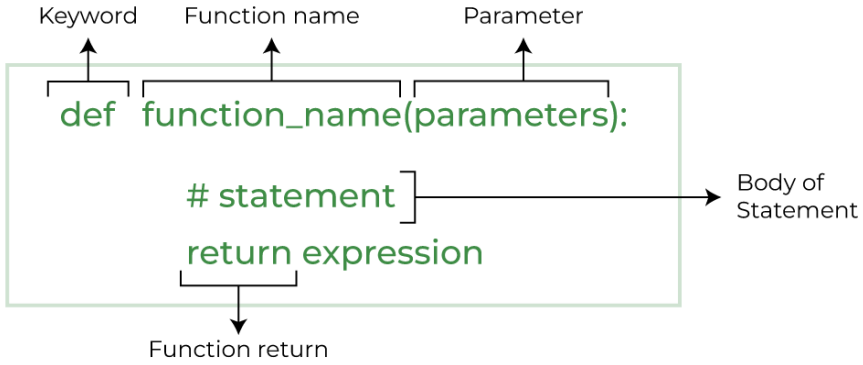
- Ease
- Less code
- More flexibility
- Tune it using arguments

When?

- Repetitive code
- Nested logic

Parts of a Function

- `def`: “This is a function”
- Name:
  - Use underscores to connect multiple words
  - Cannot begin with a number or special character
- Parameter:
  - Define things (arguments) we pass into the function
- Body:
  - Indented (“whitespace”) to indicate scope
- Return:
  - Once the code finishes, send back some value



Terms for Functions

- *Define* it using `def`
- *Return from it* when we're finished in the function
- *Call* it using its name and parentheses
- *Parameters* are what we call the things given to a function
- *Arguments* are the values we give as parameters
- We *pass* arguments to a function

```python
def example_function(print_this, specify_that):
  print("This is an example of a python function.")
  print("Everything indented under it is within its 'scope'.")

  var_a = 15

  print("For example, that variable is within its scope.")
  print("Outside this function, the variable var_a doesn't exist.")

  if var_a == 15:
    print("We can indent further, for example due to an if statement.")
    print("Here we are within the scope of both the example_function and this if statement.")

  print("We are also going to use the arguments passed to the function.")
  print(print_this)
  print("And we might want to specify {}".format(specify_that))

  print("Finally, we can return a value if we want.")
  return True

print("Now we are no longer indented and are outside the scope of our function.")
print("Using the function is known as 'calling' it:")

return_value = example_function("The first argument is one we will print", "this is the second argument")

print("The return value from the function was {}".format(return_value))
```

```python
def example_function(print_this, specify_that):
  print("This is an example of a python function.")
  print("Everything indented under it is within its 'scope'.")

  var_a = 15

  print("For example, that variable is within its scope.")
  print("Outside this function, the variable var_a doesn't exist.")

  if var_a == 15:
    print("We can indent further, for example due to an if statement.")
    print("Here we are within the scope of both the example_function and this if statement.")

  print("We are also going to use the arguments passed to the function.")
  print(print_this)
  print("And we might want to specify {}".format(specify_that))

  print("Finally, we can return a value if we want.")
  return True

print("Now we are no longer indented and are outside the scope of our function.")
print("Using the function is known as 'calling' it:")

return_value = example_function("The first argument is one we will print", "this is the second argument")

print("The return value from the function was {}".format(return_value))
```

```
Now we are no longer indented and are outside the scope of our function.
Using the function is known as 'calling' it:
This is an example of a python function.
Everything indented under it is within its 'scope'.
For example, that variable is within its scope.
Outside this function, the variable var_a doesn't exist.
We can indent further, for example due to an if statement.
Here we are within the scope of both the example_function and this if statement.
We are also going to use the arguments passed to the function.
The first argument is one we will print
And we might want to specify this is the second argument
Finally, we can return a value if we want.
The return value from the function was True
```

Parameters and Arguments

- Parameters: The names given to the things passed to a function
- Arguments: The things passed to a function

Types:

- Positional
- Defaults
- Keyword arguments

```
def evenOdd(x):
    if (x % 2 == 0):
        return "Even"
    else:
        return "Odd"

print(evenOdd(16))
print(evenOdd(7))
```

Parameters and Arguments

Positional Arguments:

- Based on the order of input to the function
- Can think of these as “required inputs”
  - If they’re missing, code will return an error

```
def example_function(print_this, specify_that):
```

```
example_function("Only passing one argument in here")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_3847/1119073431.py in <cell line: 0>()
----> 1 example_function("Only passing one argument in here")

TypeError: example_function() missing 1 required positional argument: 'specify_that'
```

```
def nameAge(name, age):
    print("Hi, I am", name)
    print("My age is ", age)
```

Parameters and Arguments

Default Arguments:

- Apply a given value to a positional argument
- If that parameter isn't provided as an argument, instead assign the variable the "default"
- *Can* have positional arguments before defaults
- *Cannot* have the reverse

```
def other_function(argument_a, argument_b, argument_c = "default for C", argument_d = 10):
```

```
other_function("AAA", "BBB")
other_function("AAA", "BBB", "CCC")
other_function("AAA", "BBB", "CCC", "DDD")
```

Parameters and Arguments

Keyword Arguments:

- Let us pass arguments in any order
  - Any argument can be called using a keyword argument
- Let us skip over defaults
- When *calling* a function, positional arguments come first, followed by keyword arguments

```
def other_function(argument_a, argument_b, argument_c = "default for C", argument_d = 10):
other_function("AAA", "BBB")
other_function("AAA", "BBB", "CCC")
other_function("AAA", "BBB", argument_d = "DDD")
other_function(argument_b = "BBB", argument_a = "AAA")
```

Parameters and Arguments

Arbitrary Arguments

- Total flexibility - nothing is named (*parameterized*) ahead of time
- List of arguments without names goes into the list *args*
- Dictionary of named arguments goes into *kwargs*

```
def myFun(*args, **kwargs):
    print("Non-Keyword Arguments (*args):")
    for arg in args:
        print(arg)

    print("\nKeyword Arguments (**kwargs):")
    for key, value in kwargs.items():
        print(f"{key} == {value}")

myFun('Hey', 'Welcome', first='to', mid='COMP', last='150')
```

```
Non-Keyword Arguments (*args):
Hey
Welcome

Keyword Arguments (**kwargs):
first == to
mid == COMP
last == 150
```

Reference and Values

One of the more confusing things in programming…

- Some things we pass to a function are changed permanently (*reference*)
- Others are passed as a copy (*value*)
- Mutable:
  - Lists, dictionaries
- Immutable:
  - Integers, strings, tuples

```
def mutable_example(x):
    x[0] = 20

lst = [10, 11, 12, 13]
mutable_example(lst)
print(lst)

def immutable_example(x):
    x = 20

a = 10
immutable_example(a)
print(a)
```

```
[20, 11, 12, 13]
10
```

Returning

- Syntax: `return` <variable>
- When a function is finished, can either just end, or can return a value
- Can happen at any point in the scope (not just at the end!)
- Once return is called, the function stops (even if more code happens later)

```
def example_function(print_this, specify_that):
  print("This is an example of a python function.")
  print("Everything indented under it is within its 'scope'.")

  var_a = 15

  print("For example, that variable is within its scope.")
  print("Outside this function, the variable var_a doesn't exist.")

  return False

  if var_a == 15:
    print("We can indent further, for example due to an if statement.")
    print("Here we are within the scope of both the example_function and this if statement.")

  print("We are also going to use the arguments passed to the function.")
  print(print_this)
  print("And we might want to specify {}".format(specify_that))

  print("Finally, we can return a value if we want.")
  return True
```

Nesting Functions

- Functions can define new functions
- Functions can call existing functions
- Functions can also self-call (recursion)

```
def example_of_multi_call():
  print("This is one function.")

def example_of_the_other_side():
  print("We can call functions from other functions.")
  example_of_multi_call()

example_of_the_other_side()
```

```
We can call functions from other functions.
This is one function.
```

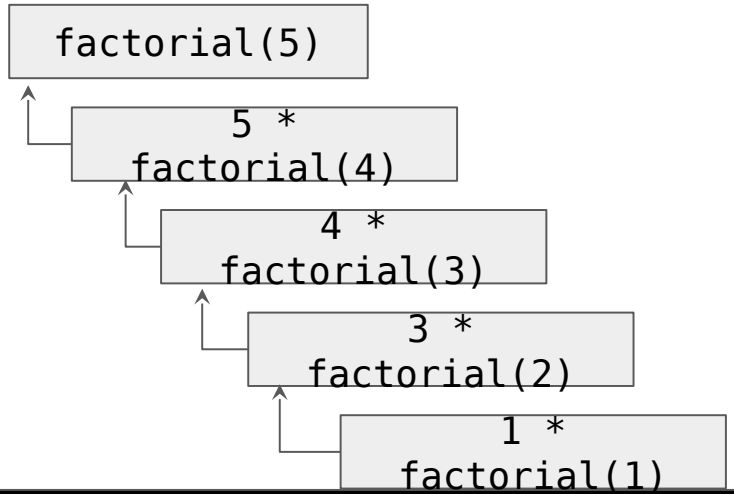
```
def nested_function(parameter_a, parameter_b):
  def internal_function(parameter_c):
    print("This function only exists in the scope of the outer function.")
    print("It CAN see the variables in the outer function!")
    print("This can be useful - and also extremely confusing.")
    print(parameter_a)

  print(parameter_b)
  internal_function("We call it like this.")

nested_function("This can be seen in both layers.", "We have to use the internal function.")
```

```
We have to use the internal function.
This function only exists in the scope of the outer function.
It CAN see the variables in the outer function!
This can be useful - and also extremely confusing.
This can be seen in both layers.
```

Recursion

- Function that calls itself
- Example: factorial

$$n! = n \times (n-1) \times (n-2) \times (n-3) \times \cdots \times 3 \times 2 \times 1$$




```python
def factorial(number):
  if number == 0:
    return 1
  else:
    return number * factorial(number - 1)

print(factorial(3))
print(factorial(10))
```

```
6
3628800
```

n	n!		
1	**1**	1	1
2	2 × **1**	= 2 × **1!**	= 2
3	3 × **2** × **1**	= 3 × **2!**	= 6
4	4 × **3** × **2** × **1**	= 4 × **3!**	= 24
5	5 × **4** × **3** × **2** × **1**	= 5 × **4!**	= 120
6	and so on	and so on	

Lab 08

Create a new notebook in Colab, name it “username_COMP150_lab08”, and write five custom functions. Make sure you call each of these functions in your notebook to test them!

1. Write a function that accepts no arguments called `test()`, and have it print “Hello, world!”
2. Write a function that accepts one positional argument, and have it print that argument. Call it `custom_print(...)`
3. Repeat `custom_print(...)` , but now give its argument a default of “Hello, world!” , and call it `custom_print_2(...)`
4. Make a function called `custom_mult()`. Have it take two arguments. Return the product of these two arguments.
5. Make a function called `using_mult()`. It should take no arguments. Have it loop five times, each time calling `custom_mult()`, to give the **square** of the numbers 0 through 4.