CIS 1100

Functions Practice, `return`, Keyword Arguments

Python

Fall 2024

University of Pennsylvania

Recap: Calling Functions with Inputs

Here is a function that takes a message and a number and prints that message that number of times.

```
def print_n_times(msg, n):
    counter = 0
    while counter < n:
        print(msg)
        counter = counter + 1
```

What happens when we call the function: `print_n_times("Hi!", 3)`?

Recap: Calling Functions with Inputs

- The function's **parameters** are `msg` and `n`.
  - These are names for variables that can be used in the body of the function
- The function call provides two **arguments**: `"Hi!"` and `3`
  - These are the values that the parameter variables will take at the start of the function execution.

```
# calling print_n_times("Hi!", 3)
def print_n_times(msg, n):
    # msg = "Hi!"
    # n = 3
    counter = 0
    while counter < n: # while counter < 3:
        print(msg)     # print("Hi!")
        counter = counter + 1
```

Activity: Counting Numbers

```python
def add_three_numbers(a, b, c):
  first_two = a + b
  last = c + first_two
  print(last)
```

- M5: calling the function as `add_three_numbers(3, 4, 7, 9)` leads the program to immediately crash
- M6: calling the function as `add_three_numbers("three", "four", "five")` leads the program to immediately crash

A: True, B: False

Activity: Working Towards Writing a Function

Assuming you have a list `lst` containing a bunch of numbers, write a couple of loops that print out all of the **negative** numbers and then all of the **non-negative numbers**. (C12, but leave just a little space at the top)

e.g.

```
lst = [9, -19, 31, -13, 1, 2]
# TODO: Your loop(s) here
```

🖨 👇

```
-19 -13 9 31 1 2
```

You're not writing a whole function yet! Just write some lines & loops like you've been doing before.

Activity: Working Towards Writing a Function

Write the signature for a function that prints out all of the **negative** numbers and then all of the **non-negative numbers**. (L11)

Remember: a signature consists of a `def`*, a function name, and a list of parameters the function should be called with.*

Activity: Working Towards Writing a Function

Add a signature to the code you wrote for (C12) in order to turn it into a function that can be called.

Then, in (L13), write an example of a function call that would print out the following output:

```
-30 -14 3 19 8
```

New: `return`

Function calls are themselves *expressions*, meaning that they always have a value.

- The value of a function call is determined by the value that function **returns**

`return` is keyword that serves two purposes:

- stops function execution in its tracks
- provides a value for the expression of the function call

`return`: An Example

```
def multiply_two_numbers(a, b):
    print(f"Multiplying {a} x {b}!")
    product = a * b
    return product
```

If we write the call `multiply_two_numbers(3, 7)`, then...

```
    # a = 3
    # b = 7
    print(f"Multiplying {a} x {b}!")
    product = a * b                     # product = 3 * 7
    return product                      # return 21
```

...we return the value of `product`, which is `21` based on this function call. The following therefore evaluates to `True`:

```
multiply_two_numbers(3, 7) == 21
```

Printing vs. Returning

An output that's *printed* is not the same as an output that's *returned.*

- Any call to `print()` will make text appear on the screen, but it doesn't produce a value
- If a function is supposed to calculate and create some value (e.g. the product of two numbers), it must *return* that value in the function body.

Functions that Have No `return`

```python
def our_min(lst):
  smallest = lst[0]
  for elem in lst:
    if elem < smallest:
      smallest = elem
  print(smallest)

def our_len(lst):
  running_sum = 0
  for elem in lst:
    running_sum += 1
  print(running_sum)
```

```python
some_numbers = [1000, 3, 8]

result = our_min(some_numbers) # 🖨➡ ??
print(result)               # 🖨➡ ???

result = our_len(some_numbers) # 🖨➡ ??
print(result)               # 🖨➡ ???
```

These functions both *compute* some value and then *print* it but do not *return* it.

Functions that Have No `return`

```python
def our_min(lst):
  smallest = lst[0]
  for elem in lst:
    if elem < smallest:
      smallest = elem
  print(smallest)

def our_len(lst):
  running_sum = 0
  for elem in lst:
    running_sum += 1
  print(running_sum)
```

```python
some_numbers = [1000, 3, 8]

result = our_min(some_numbers) # 🖨 ➡ 3
print(result)                  # 🖨 ➡ None

result = our_len(some_numbers) # 🖨 ➡ 3
print(result)                  # 🖨 ➡ None
```

These functions both *compute* some value and then *print* it but do not *return* it.

Adding `return`

```python
def our_min(lst):
  smallest = lst[0]
  for elem in lst:
    if elem < smallest:
      smallest = elem
  return smallest)

def our_len(lst):
  running_sum = 0
  for elem in lst:
    running_sum += 1
  return running_sum)
```

```python
some_numbers = [1000, 3, 8]

result = our_min(some_numbers) # 🖨️➡️ ???
print(result)                  # 🖨️➡️ ???

result = our_len(some_numbers) # 🖨️➡️ ???
print(result)                  # 🖨️➡️ ???
```

These functions now *compute* some value and then *return* it but do not *print* it.

Adding `return`

```python
def our_min(lst):
  smallest = lst[0]
  for elem in lst:
    if elem < smallest:
      smallest = elem
  return smallest)

def our_len(lst):
  running_sum = 0
  for elem in lst:
    running_sum += 1
  return running_sum)
```

```python
some_numbers = [1000, 3, 8]

result = our_min(some_numbers) # 🖨️➡️ Nothing!
print(result)                  # 🖨️➡️ 3

result = our_len(some_numbers) # 🖨️➡️ Nothing!
print(result)                  # 🖨️➡️ 3
```

These functions now *compute* some value and then *return* it but do not *print* it.

The Point of No `return`?

`return` works as a stopping/exit point for your program. If you execute a line with `return`, you will leave that function call execution.

```python
def print_all_above(lst, k):
  for elem in lst:
    if elem > k:
      print(elem)

print_all_above([5, 10, 15], 8)
```

🖨️ 👇

```
10 15
```

The Point of No `return`?

`return` works as a stopping/exit point for your program. If you execute a line with `return`, you will leave that function call execution.

```python
def print_first_above(lst, k):
  for elem in lst:
    if elem > k:
      print(elem)
      return

print_all_above([5, 10, 15], 8)
```

🖨️ 👇

```
10
```

The Point of No `return`?

`return` works as a stopping/exit point for your program. If you execute a line with `return`, you will leave that function call execution.

```
def return_first_above(lst, k):
  for elem in lst:
    if elem > k:
      return elem

print_all_above([5, 10, 15], 8)
```

🖨️ 👇

```

```

...but it does return `10`!

Activity

```python
def foo(l):
    for i, n in enumerate(l):
        if n == i:
            return n
        if n == len(l):
            print("🕐")
    print("🚩")
```

- What is the value of `x` if we run `x = foo([3, 1, 4])`? **(S7)**
- What values are printed if we run `x = foo([3, 1, 4])`? **(S8)**
- What is the value of `x` if we run `x = foo([10, 11, 12])`? **(S9)**
- What values are printed if we run `x = foo([3, 1, 4])`? **(S10)**

Keyword Arguments

Sometimes we want our functions to be able to take *default values* for their inputs. We can do this with **keyword arguments.**

```python
def divide(a, b, rounding=False):
    result = a / b
    if rounding:
        return round(result)
    else:
        return result
```

`rounding` is a keyword argument that is defined by its *name* as well as the *default value* that it takes if it is not replaced.

Keyword Arguments

```
def divide(a, b, rounding=False):
    result = a / b
    if rounding:
        return round(result)
    else:
        return result
```

We can do any of the following:

```
>>> divide(3422, 194)
17.63917525773196
>>> divide(3422, 194, rounding=True)
18
>>> divide(3422, 194, True)
18
>>> divide(3422, 194, False)
17.63917525773196
```

Rules of Keyword Arguments

Signatures:

- All keyword parameters have to be provided AFTER all the positional ones
- A keyword parameter is defined by writing `identifier=<default_value>`
- Can have as many as you want, including ONLY keyword parameters

Calls:

- All keyword arguments have to be passed in AFTER all positional inputs, but from there can be in any order
- Keyword arguments can be given positionally or by name, but you should always just give thme by name

Good or Bad?

```python
def fun(a, b, c=13, d):
    pass
```

Good or Bad?

```python
def fun(a, b, c=13, d):
    pass
```

BAD!

Good or Bad?

```python
def fun(a=13, n="haha"):
    pass
```

Good or Bad?

```python
def fun(a=13, n="haha"):
    pass
```

GOOD!

Good or Bad?

```python
def fun(a, b, c=, d=13):
    pass
```

BAD!

Good or Bad?

```python
def fun(x, y, z=0):
    pass
```

Then,

```python
...
fun(3, 4, 0)
...
```

Good or Bad?

```
def fun(x, y, z=0):
    pass
```

Then,

```
...
fun(3, 4, 0)
...
```

OK, but redundant!

Good or Bad?

```python
def fun(x, y, z=0):
    pass
```

Then,

```python
...
fun(z=0, 3, 4)
...
```

Bad!

Good or Bad?

```python
def fun(x, y, z=0):
    pass
```

Then,

```python
...
fun(3, 4, z=x+y)
...
```

Bad!

Good or Bad?

```python
def fun(x, y, z=0):
    pass
```

Then,

```python
...
fun(3, 4)
...
```

Good!