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
```

Can you tell what *types* the parameters are for this function from its signature and body?

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
  - Based on this function, we still don't know
    what the appropriate types are as arguments.

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

Activity: Valid Arguments

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

A: True, B: False, these function calls crash the program

- M1: `print_n_times("hi", 1.2)`
- M2: `print_n_times(4.1, 1.1)`
- M3: `print_n_times(["hi"], 10)`

Python Type Hints

A function written this way makes it difficult for the programmer to determine the expected argument types, such as *str*, *int*, *float*, *list*, *range*, and so on.

To make it easier on us, let's go ahead and use type hints!

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

Type hints specify the expected parameter types, ensuring the function behaves as intended.

Python Type Hints

```
def print_n_times(msg: str, n: int):
```

- `msg: str`
  - tells us that the parameter msg *should* be a string
- `n: int`
  - tells us that the parameter n *should* be an integer

Note: type hints are not **enforced** when running a program. Even with type hints we could run `print_n_times([1, 2, 3], 10)`.

Type Hints are just for us to know how to correctly use a function.

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

If we 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
```

`return` : An Example with type hints!

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

This function signature tells us....

```
def multiply_two_numbers(a: int, b: int) -> int:
```

- `a` and `b` are integers
- `-> int:` tells us that the function returns an integer!

That way we don't have to wonder if it does return a value or not.

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`

```
def our_min(lst: list): # no -> needed
  smallest = lst[0]
  for elem in lst:
    if elem < smallest:
      smallest = elem
  print(smallest)

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

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

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

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

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

Adding `return`

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

def our_len(lst: list[int]) -> int:
    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: list[int]) -> int:
    smallest = lst[0]
    for elem in lst:
        if elem < smallest:
            smallest = elem
    return smallest

def our_len(lst: list[int]) -> int:
    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.

```
def print_all_above(lst: list[int], k: int):
  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.

```
def print_first_above(lst: list[int], k: int):
  for elem in lst:
    if elem > k:
      print(elem)
      return

print_first_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: list[int], k: int) -> int:
  for elem in lst:
    if elem > k:
      return elem

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

🖨️ 👇

```

```

...but it does return `10`!

Activity

```python
def foo(curr_list: list[int]):
    for index, num in enumerate(curr_list):
        if num == index:
            return num
        if num == len(curr_list):
            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([10, 11, 12])`? **(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.**

```
def divide(a: int, b: int, rounding: bool = False ) -> float:
    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: int, b: int, rounding: bool = False) -> float:
    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
```

note: type hints to support returning either an int or float are possible...

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 them by name

Keyword Arguments: Valid or Invalid?

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

Keyword Arguments: Valid or Invalid?

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

Invalid!

Keyword Arguments: Valid or Invalid?

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

Keyword Arguments: Valid or Invalid?

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

Valid!

Keyword Arguments: Valid or Invalid?

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

Keyword Arguments: Valid or Invalid?

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

Invalid!

Keyword Arguments: Valid or Invalid?

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

fun(3, 4)
```

Keyword Arguments: Valid or Invalid?

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

fun(3, 4)
```

Valid!

Keyword Arguments: Valid or Invalid?

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

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

Keyword Arguments: Valid or Invalid?

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

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

Valid (but redundant!)

Keyword Arguments: Valid or Invalid?

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

fun(z = 0, 2, 3)
```

Keyword Arguments: Valid or Invalid?

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

fun(z = 0, 2, 3)
```

INVALID!!!!!!!!!!!!!

```
SyntaxError: positional argument follows keyword argument
```

Keyword Arguments: Valid or Invalid?

```
def fun(x, y, z=0):
    return x + y * z

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

Keyword Arguments: Valid or Invalid?

```
def fun(x, y, z=0):
    return x + y * z

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

Invalid!!!

Keyword Arguments: Valid or Invalid?

```
def fun(x, y, z=0):
    return x + y * z

fun(3, 4)
```

Valid!

Lecture Activity L11

```python
import random
def mystery_function(param_one: int, param_two: int, mystery: bool = True) -> float:
    if mystery:
        param_one = random.random() * 100
        param_two = random.random() * 100

    result = (param_one + param_two) / 2

    return float(result)
```

What does this function do and what is the mystery keyword *controlling* here*?

Lecture Activity C12

```python
def mystery_function(param_one: str, param_two: str, param_three: str = "don't stop") -> float:
    if "stop" in param_one.lower() or "stop" in param_two.lower() or "stop" in param_three.lower():
        return float('nan') #Not a Number

    total_length = len(param_one) + len(param_two) + len(param_three)
    result = total_length / 3

    return result
```

What does this function do? Under what conditions does it "fail"