COMP 364: Functions II

Carlos G. Oliver, Christopher Cameron

October 2, 2017

Outline

1. Recap + Warmup
2. Functions Theory: Positional vs Keyword Arguments
3. Functions practice: Namespaces.
4. Practice Problem

Recap

- Anatomy of a function
- Importing

Warm-up

- Write a function called `splice()` that takes two strings `dna1, dna2` and four integers `x, y, a, b`. The function returns a new string slices defined by the four integers pasted together.

```
>>> splice("AAACCGGG", "GGTATACCG", 0, 3, 2, 5)
AAATA
```

Positional Arguments

- Positional arguments are **mandatory** and must be passed in order.
- i.e. their position in the function call matters.

```
1 import numpy as np
2 #function with only positional arguments
3 def my_subtract(a, b):
4     return a - b
5 my_subtract(1, 4) != my_subtract(4, 1)
6 my_subtract(1) # ERROR
```

Keyword Arguments

- Keyword arguments must be passed after all positional arguments and are **optional**.
- Syntax: `def func_name(pos1, pos2, ...,pos3, kwarg1=a, kwarg2=b, ..)`
- You can specify keyword arguments in any order you like.

```
1  def my_calc(a,b, operation="add"):
2      if operation == "add":
3          return a + b
4      elif operation == "subtract":
5          return a - b
6      elif operation == "multiply":
7          return a * b
8      elif operation == "divide":
9          return a / b
10     else:
11         print("incorrect operation string")
12         return None
```

Keyword Arguments

- Keyword arguments let your function have some default value for an object while letting the caller change it if necessary.
- Lets the user optionally control how the function behaves.

```
x = 5
y = 7
adding = my_calc(x, y)
difference = my_calc(x, y, operation="subtract")
product = my_calc(x, y, operation="multiply")
```

Arbitrary number of arguments

- Python lets you define functions that take an arbitrary number of positional or keyword arguments using the `*args` and `**kwargs` conventions respectively.
- For this class we won't be making too much use of this but it's good to be aware of it.
- Say I want a function that adds any number of numbers together

```
1  def my_sum(*args):
2      tot = 0
3      for a in args:
4          tot = tot + a
5      return tot
```

- `args` stores the positional arguments as a tuple
- The `*` tells python that this will contain positional arguments.

Arbitrary number of arguments

- We can do the same for keyword arguments.
- The ** tells python that this name will contain keyword arguments

```
1 def class_grades(**kwargs):
2     for k in kwargs:
3         print(f"{k}'s GPA is {kwargs[k]}")
4 class_grades(tina=3.4, bob=2.5, jerry=4.0,
  ↪  maureen=3.0)
```

- Keyword arguments are stored in what is called a dictionary. More on that later. For now think of it as a list that you can access by name instead of index.

Combining arguments

- You can combine all the argument types. Remember that we keep mandatory positional arguments first, then the rest.

```
1 #the most general function, takes anything as input
2 def foo(*args, **args):
3 #takes two mandatory posiitonal arguments and then
   ↪ anything
4 def faa(tim, robert, *args, **args)
5 #takes no positional arguments but sets a default
   ↪ value for keyword dog
6 def faa(animal="dog")
```

Practice

- Write a function called `my_seq()` that accepts a sequence of nucleotides as a string (mandatory). The function also takes a keyword argument `trim`, `direction` which trims the value of `trim` nucleotides from the specified `direction`. The default values for `trim` is 0 and for `direction` is `"front"`.

```
1 newseq = my_seq("AAACGGG", trim=3, direction="back")
   ↪  #"AAAC"
2 newseq = my_seq("AAACGGG") #"AAACGGG"
```

Function namespaces

- What happens if I run this code?

```
1 juliet = "Wherefore art thou Romeo?"
2 def foo():
3     print(juliet)
4     romeo = "I'm the VIP room, Juliet."
5 foo()
6 print(romeo) #ERROR
```

- A function in Python has **its own namespace**!
- If a Python program is a party, a function is a VIP room where the people in the room can see the rest of the party but the rest of the party can't see the VIP.

Namespaces

- When trying to access a `name`, python looks first in your local namespace, if it doesn't find it it looks up.
- Local → Global/Module → Built-in
- FYI: Module is just another name for a Python file.

```
1 id(id) # built-in
2 a = "bob" # global/module
3 z = "hi"
4 def foo():
5     b = "mary" # local
6     z = "tim"
```

1

[1] https://www.programiz.com/python-programming/namespace

Example #1

```
1  show = "Game of Thrones"
2
3  def switch_show():
4      show = "Narcos"
5  print(show)
6  switch_show(show)
7  print(show)
```

Example #1: Solution

```
1  show = "Game of Thrones"
2
3  def switch_show():
4      show = "Narcos"
5  print(show)
6  switch_show(show)
7  print(show)
```

- We print `"Game of Thrones"`
- The name binding on line 4 lives only inside the function namespace.

Example #2

```
1  def switch_show():
2      show = "Narcos"
3  def switch_movie():
4      movie = "Harry Potter"
5      print(show)
6  switch_show()
7  switch_movie()
```

Example #2: Solution

```
1 def switch_show():
2     show = "Narcos"
3 def switch_movie():
4     movie = "Harry Potter"
5     print(show)
6 switch_show()
7 switch_movie()
```

- ERROR. `switch_movie()` does not have access to the names in `switch_show()`

Example #3

```
1  def switch_show():
2      show = "Narcos"
3      def switch_movie():
4          movie = "Harry Potter"
5          print(show)
6      switch_movie()
7      print(movie)
8  switch_show()
```

Example #3: Solution

```
1  def switch_show():
2      show = "Narcos"
3      def switch_movie():
4          movie = "Harry Potter"
5          print(show)
6      switch_movie()
7      print(movie)
8  switch_show()
```

- ERROR on line 7.
- The name `movie` is not accessible from a higher level.

Function arguments and Namespaces

- What happens when we pass an argument to a function?

```
1 def my_func(func_arg):
2     print(id(func_arg))
3 current_name = "Hello"
4 my_func(current_name)
```

- Python binds the name `func_arg` to the object with the name `current_name`
- The name `func_arg` now lives inside the function's namespace.

Example #1

```
1  def winter_is_here(season):
2      season = "Winter"
3      print(f"inside function: {season}")
4  season = "Fall"
5  winter_is_here(season)
6  print(f"in global scope {season}")
```

Example #1: Solution

```
1 def winter_is_here(season):
2     season = "Winter"
3     print(f"inside function: {season}")
4 season = "Fall"
5 winter_is_here(season)
6 print(f"in global scope: {season}")
```

```
1 "inside function: Winter"
2 "in global scope: Fall"
```

- The function has its own `season` name bound to the same object as the external `season` name. But in line 2 it makes a new binding to the object `"Winter"`. This binding is not seen in the global scope so the value of the external object doesn't change.

Example # 2

```
1 def add_season(s):
2     seasons.append(s)
3     seasons = False
4 seasons = []
5 add_seasons("Winter")
6 print(seasons)
```

Example # 2: Solution

```
1 def add_season(s):
2     seasons.append(s)
3     seasons = False
4 seasons = []
5 add_seasons("Winter")
6 print(seasons)
```

- We print `["Winter"]`
- The function looks for `seasons` in its namespace and finds no binding. So it looks up one level and finds a binding on line 4. Since lists are mutable it can modify it by adding a value.
- Line 3 is a binding *inside* the function so it does not affect `seasons` in the global scope.

Reminder: Mutability

- Immutable types: we can't change the value of an object
  - `str, int, float, bool, tuple`
- Mutalbe types: we can change the value while keeping the same object.
  - `list`

```
>>> lala = ["steve", "rick"]
>>> id(lala)
4415750216
>>> lala.append("mary")
>>> id(lala)
4415750216
>>> print(lala)
["steve", "rick", "mary]
>>> s = "roger"
>>> s[1] = "R" #ERROR, strings are immutable
```

Namespace rules

1. Scope: You can access name bindings in higher namespaces but not lower or equal depth.
2. Mutability: Always remember the difference between mutable and immutable objects.

 [2]

[2] https://fthmb.tqn.com

Advanced function tricks: `lambda`

- You can use the keyword `lambda` to create a nameless function as an expression.
- Syntax: `lambda args: expression`

```
1  #bind the name ‘a‘ to the lambda expression
2  >>> a = lambda x, c: x*x + c
3  >>> a(2, 1)
4  5
```

Advanced function tricks: `map()`

- The `map(func, s)` function applies `func` to every item in `s`

```
>>> f = lambda x: x+1
>>> map(a, [1, 2, 3, 4])
[2, 3, 4, 5]
```

- `map()` works on functions that only take one argument. You can go around this by packing arguments inside a tuple.

```
>>> g = lambda x: x[0] + x[1]
>>> map(g, [(1, 1), (2, 2), (3, 4)])
[2, 4, 7]
```

Practice

- Write a function called `transcript` that takes two positional arguments: `name`, `term` and any number of keyword arguments. Each keyword argument indicates the name of a class and its value is your percentage grade for that class. The function prints a "report card" with the letter grade for each class, your average GPA and returns the average.

```
>>> my_avg = transcript("Carlos", Fall 2017",
 ↪  COMP364=89, MATH324=55, BIOL200=66)
Name: Carlos
Term: Fall 2017
COMP364: A
MATH324: F
BIOL200:B-

GPA:3.1
>>> print(my_avg)
3.1
```