Outline

1. Recap + Warmup
2. Functions Theory: Positional vs Keyword Arguments
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.

```python
>>> 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.

```python
import numpy as np

#function with only positional arguments
def my_subtract(a, b):
    return a - b

my_subtract(1, 4) != my_subtract(4, 1)
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.

```python
def my_calc(a, b, operation="add"):
    if operation == "add":
        return a + b
    elif operation == "subtract":
        return a - b
    elif operation == "multiply":
        return a * b
    elif operation == "divide":
        return a / b
    else:
        print("incorrect operation string")
        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.

```python
1 x = 5
2 y = 7
3 adding = my_calc(x, y)
4 difference = my_calc(x, y, operation="subtract")
5 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

```python
def my_sum(*args):
    tot = 0
    for a in args:
        tot = tot + a
    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

```python
def class_grades(**kwargs):
    for k in kwargs:
        print(f"{k}'s GPA is {kwargs[k]}")
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.

```python
#the most general function, takes anything as input
def foo(*args, **args):
#takes two mandatory positional arguments and then
   anything
def faa(tim, robert, *args, **args):
#takes no positional arguments but sets a default
   value for keyword dog
def faa(animals="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".

```python
def my_seq(sequence, trim=0, direction="front"):
    # Implementation

newseq = my_seq("AAACG GG", trim=3, direction="back")
# "AAAC"
newseq = my_seq("AAACG GG")
# "AAACG GG"
```
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.

```python
id(id)    # built-in
a = "bob"  # global/module
z = "hi"
def foo():
    b = "mary"  # local
    z = "tim"
```

show = "Game of Thrones"

def switch_show():
    show = "Narcos"
print(show)
switch_show(show)
print(show)
Example #1: Solution

```python
show = "Game of Thrones"

def switch_show():
    show = "Narcos"
print(show)
switch_show(show)
print(show)
```

- We print "Game of Thrones"
- The name binding on line 4 lives only inside the function namespace.
Example #2

def switch_show():
    show = "Narcos"
def switch_movie():
    movie = "Harry Potter"
    print(show)
switch_show()
switch_movie()
Example #2: Solution

```python
def switch_show():
    show = "Narcos"

def switch_movie():
    movie = "Harry Potter"
    print(show)

switch_show()
switch_movie()
```

- ERROR. `switch_movie()` does not have access to the names in `switch_show()`
Example #3

def switch_show():
    show = "Narcos"

def switch_movie():
    movie = "Harry Potter"

    print(show)
    switch_movie()
    print(movie)

switch_show()
Example #3: Solution

```python
def switch_show():
    show = "Narcos"

def switch_movie():
    movie = "Harry Potter"
    print(show)
    switch_movie()
    print(movie)
    switch_show()
```

- ERROR on line 7.
- The name `movie` is not accessible from a higher level.
What happens when we pass an argument to a function?

```python
def my_func(func_arg):
    print(id(func_arg))
    current_name = "Hello"
    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.
def winter_is_here(season):
    season = "Winter"
    print(f"inside function: {season}\n")
season = "Fall"
winter_is_here(season)
print(f"in global scope {season}"
Example #1: Solution

```python
def winter_is_here(season):
    season = "Winter"
    print(f"inside function: {season}\")
    season = "Fall"
winter_is_here(season)
print(f"in global scope: {season}\")
```

"inside function: Winter"
"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

```python
def add_season(s):
    seasons.append(s)
    seasons = False
seasons = []
add_seasons("Winter")
print(seasons)
```
def add_season(s):
    seasons.append(s)
    seasons = False

seasons = []

add_seasons("Winter")

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
- Mutable types: we can change the value while keeping the same object.
  - list

```python
>>> 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.

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`

```
#bind the name ‘a’ to the lambda expression
>>> a = lambda x, c: x*x + c
>>> a(2, 1)
5
```
Advanced function tricks: `map()`

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

```python
>>> f = lambda x: x+1
>>> map(f, [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.

```python
>>> 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.

```python
>>> 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
```