CS 1112: Introduction To Programming

Function Arguments

Named Arguments, Positional Arguments, and More

Dr. Nada Basit // basit[*at*]Virginia[*dot*]edu

Spring 2024

Friendly Reminders

- Your ***safety*** and ***comfort*** is important!
  - If you choose to wear a mask you are welcome to do so
  - *We will interpret wearing a mask as being **considerate and caring** of others in the classroom (not that you are sick), and realize that some may choose to mask to **remain distanced***
- Be an ***active*** participant in your learning!
  You're welcome and ***encouraged*** to ask questions during class!
- If you feel ***unwell***, or think you are, please stay home
  - *We will work with you!*
  - Get some rest ☺
  - View the recorded lectures – *please allow 24-48 hours to post*
  - *Contact us!*

Announcements

- **Quiz 3** has been graded! Scores can be seen on Sherlock & Canvas.
- **Quiz 4** will be released this afternoon and is due by 11:00pm on Monday (***2/19***)!
  - No late quizzes accepted
  - No make-up quizzes allowed
  - If you believe your computer is glitching, it's a good idea to copy down your answers to each of the questions in a word document. In the event something happens, you can send me your solutions.
  - *Note: in general, will **cannot and will not** accept quiz solutions via **email**. We will only accept them in the case where your quiz may have glitched and we no longer have your submitted answers.*
  - **Take quiz on:** Sherlock.cs.virginia.edu
- **PA03 - Functions** is out and is due by 11:00pm on Wednesday (***2/21***)!
  - ***Start early!!*** Submit on Gradescope: your .py file
- *Coming up…*
  - **Exam 1** on February 28, 2024 (during class time)

What happens when you invoke (call) a function

- Python creates **memory** for the function (we don't see this)
- Argument values are assigned to parameter names
- Lines of code in the body of the function are **executed**
- When the function encounters a return statement, a value is passed back to the **caller**, and the function ends
- Python **removes the memory** for the function *(all values that existed only in the function are gone)*

Order of evaluation in this example:
Line 1 (skip to end of function)
-> Line 5 -> Line 6
-> Line 7 (function is called)
    -> Line 1 -> Line 2
    -> Line 3 (return statement)
-> Line 7 -> Line 8 -> Line 9

```
1:  def add_stuff(a):
2:    b = a + 5
3:    return b
4:
5:  b = 4
6:  print("Statement 1: The value of b is", b)
7:  y = add_stuff(7)
8:  print("Statement 2: The value of y is", y)
9:  print("Statement 3: The value of b is", b)
```

Named Arguments and Default Parameters Values

Named Arguments and Default Parameters Values

Usually, the number or arguments needs to **match** the number of parameters

A **default** gives us the option of providing an argument for a certain parameter

If we don't provide an argument for that parameter, the function uses the default argument instead

Named Arguments and Default Parameter Values

- Consider the function declaration below:
```
def my_function(a, b=15, c="dog"):
```

- When calling the function above, b and c have *default* values
  - This means when you call the function, you can ***choose*** to either specify b and c or use the default variables

- Examples:
  - `my_function(3, 4, "dog")` - a is 3, b is 4, c is “dog”
  - `my_function(7)`- a is 7, b and c take the *default* values of 15 and “dog” respectively
  - `my_function(7, c="desk")`- a is 7, c is “desk”, b takes on the default value of 15

Q1: Named Arguments

What is printed?

```
def add_stuff(x, y=5):
  x = x + 3
  return x + y

print(add_stuff(10))
```

Q1: Named Arguments

What is printed?

18

```
def add_stuff(x, y=5):
  x = x + 3
  return x + y

print(add_stuff(10))
```

Q2: Vocabulary check-in

```
def find_hypotenuse(c, d):
```

c ?

Q2: Vocabulary check-in

```
def find_hypotenuse(c, d):
```

parameter

Q3: Scope

Are there two different variables named **x**, or just one?

```
def add_stuff(y):
    x = y + 5
    return x

x = 4
r = add_stuff(10)
print(x)
```

Q3: Scope

Are there two different variables named **x**, or just one?

Two different **x**'s

```
def add_stuff(y):
    x = y + 5
    return x

x = 4
r = add_stuff(10)
print(x)
```

Q4: Scope

What is printed?

```
def add_stuff(y):
    x = y + 5
    return x

x = 4
r = add_stuff(10)
print(x)
```

Q4: Scope

What is printed?

4

(x = 10 + 5 inside…)
(Global x outside remains 4!)

```
def add_stuff(y):
    x = y + 5
    return x

x = 4
r = add_stuff(10)
print(x)
```

```python
# acceleration due to gravity
# given time, calculate distance
def get_distance(time):
   position = ((9.81 * (time**2))/2) + (0 * time)
   return position

def get_distance2(time, accel):
   position = ((accel * (time**2))/2) + (0 * time)
   return position

def get_distance3(time, accel, init_vel):
   position = ((accel * (time**2))/2) + (init_vel * time)
   return position


def get_distance4(time, accel=9.81, init_vel=0.0):
   position = ((accel * (time**2))/2) + (init_vel * time)
   return position

seconds = 1

print(get_distance4(seconds))
print(get_distance4(seconds, 4.0))
print(get_distance4(seconds, init_vel=2))  # keyword parameter
print(get_distance4(seconds, 4.0, 2))

# #seconds = int(input('How many seconds has the object fallen? '))
# print(get_distance(seconds))
# # print(get_distance2(seconds, 4.0))
# print(get_distance3(seconds, 9.81, 0.0))
# print(get_distance3(seconds, 4.0, 1.0))
```

When you are **not providing all the arguments**, to ensure no confusion, name the parameter.

Here, we are providing seconds (that relates to the first parameter), and init_vel (that relates to the third parameter). If we only put (seconds, 2), then it would assume accel was 2!

```python
# Let's create a function that can compute a tip for us. Based on the
# amount we were charged for our meal at a restaurant, compute the
# amount that we should leave as a tip.
def find_total_bill(food_amount, tip_percent=.20, flat=1.00):
  tip = (food_amount * tip_percent) + flat
  return(food_amount + tip)



bill_amount = float(input("How much is your bill? "))
percent = float(input("What percent do you want to tip? "))
fee = float(input("What is the flat fee amount? "))

print("You should pay", str(find_total_bill(bill_amount, percent, fee)))
```

```python
#  In this example, the find_tip function has 3 parameters. The last 2
#  parameters each have default values assigned to them. When this
# function is called, arguments do not need to be sent for parameters
# that already have default values assigned. But if arguments are sent
# in, they will be assigned to these parameters instead of using the
# default values.
def find_tip(bill, percentage=.20, msg="That was excellent service"):
  amount_of_tip = bill * percentage
  print(msg)
  return amount_of_tip


bill_amount = float(input("How much was your bill? "))
#  Only 1 argument is sent. The last 2 parameters will be their defaults
tip_amount = find_tip(bill_amount)
print("For a bill of", bill_amount, "tip", round(tip_amount, 2))

#  The same function is called, but here 3 arguments are sent
customized_tip_amount = find_tip(bill_amount, .45, "THANKS!!!!!!!!")
print("For a bill of", bill_amount, "your customized tip",
round(customized_tip_amount, 2))

#  When calling a function that has some default arguments, the
#  arguments that have defaults can be skipped and later ones
#  assigned by using the
#  parameter names of the ones that you want to assign. In this example,
#  the 2nd argument, percentage is skipped and the parameter msg is
#  assigned by name
find_tip(4, msg="This is a new message")
print("For a bill of", bill_amount, "your customized tip",
round(customized_tip_amount, 2))
```

Positional Arguments vs. Named (Keyword) Arguments

Positional Arguments vs. Named (Keyword) Arguments

- Arguments: when you **call** a functions
  - arguments are either ***positional arguments*** or ***named arguments***
- Parameters: when you **define** a function
- parameters are either ***required parameters*** or ***optional parameter***

```
def my_function(a, b = 15, c = "cat"):
```

required parameter

optional parameter

Positional Arguments vs. Named (Keyword) Arguments

- Arguments: when you **call** a functions
  - arguments are either ***positional arguments*** or ***named arguments***
- Parameters: when you **define** a function
- parameters are either ***required parameters*** or ***optional parameter***

```
def my_function(a, b = 15, c = "cat"):
```

```
my_function(10)
my_function(a=10)
```

named argument

positional argument

Note: These two lines do the same thing.

Practice calling “roll_2_dice” function in different ways…!

- Notice the optional parameter of “roll_2_dice” function.

```
import random

def roll_2_dice(num_sides=6):
    dice1_roll = random.randint(1, num_sides)
    dice2_roll = random.randint(1, num_sides)
    return dice1_roll + dice2_roll

print(roll_2_dice())
print(roll_2_dice(4))
print(roll_2_dice(num_sides=4))
```

Practice calling “roll_2_dice” function in different ways…!

- Notice the optional parameter of “roll_2_dice” function.

```
import random

def roll_2_dice(num_sides=6):
    dice1_roll = random.randint(1, num_sides)
    dice2_roll = random.randint(1, num_sides)
    return dice1_roll + dice2_roll

print(roll_2_dice())
print(roll_2_dice(4))
print(roll_2_dice(num_sides=4))
```

What is the value of num_sides?	*Answer?*
What is the value of num_sides?	*Answer?*
What is the value of num_sides?	*Answer?*

Practice calling “roll_2_dice” function in different ways…!

- Notice the optional parameter of “roll_2_dice” function.

```
import random

def roll_2_dice(num_sides=6):
    dice1_roll = random.randint(1, num_sides)
    dice2_roll = random.randint(1, num_sides)
    return dice1_roll + dice2_roll

print(roll_2_dice())
print(roll_2_dice(4))
print(roll_2_dice(num_sides=4))
```

What is the value of num_sides?	**6**
What is the value of num_sides?	**4**
What is the value of num_sides?	**4**

Activity - Make your own function header

- Make a function based on a story or your daily life
- One ***required*** parameter
- Two ***optional*** parameters

For example...

- **teach_class**
  - number of students: no default
  - class_name: "CS1112"
  - time: "2 PM"

Activity - Make your own function header

- Make a function based on a story or your daily life
- One ***required*** parameter
- Two ***optional*** parameters

For example...

- **teach_class**
  - number of students: no default
  - class_name: "CS1112"
  - time: "2 PM"

```
def teach_class(num_students, name="CS1112", time="2 PM"):
```

Activity - Use your own function :)

```
def teach_class(num_students, name="CS1112", time="2 PM"):
```

What will the parameter values be if you... call the function with *only one positional argument*

```
teach_class(30)
```

What will the parameter values be if you... call the function with *two positional arguments*

```
teach_class(40, "CS2100")
```

What will the parameter values be if you... call the function with *no arguments*

```
teach_class()
```

Activity - Use your own function :)

```
def teach_class(num_students, name="CS1112", time="2 PM"):
```

What will the parameter values be if you... call the function with *only one positional argument*

```
teach_class(30)
```

num_students=30, name="CS1112", time="2 PM"

What will the parameter values be if you... call the function with *two positional arguments*

```
teach_class(40, "CS2100")
```

What will the parameter values be if you... call the function with *no arguments*

```
teach_class()
```

Activity - Use your own function :)

```
def teach_class(num_students, name="CS1112", time="2 PM"):
```

What will the parameter values be if you... call the function with *only one positional argument*

```
teach_class(30)
```

num_students=30, name="CS1112", time="2 PM"

What will the parameter values be if you... call the function with *two positional arguments*

```
teach_class(40, "CS2100")
```

num_students=40, name="CS2100", time="2 PM"

What will the parameter values be if you... call the function with *no arguments*

```
teach_class()
```

Activity - Use your own function :)

```
def teach_class(num_students, name="CS1112", time="2 PM"):
```

What will the parameter values be if you... call the function with *only one positional argument*

```
teach_class(30)
```

num_students=30, name="CS1112", time="2 PM"

What will the parameter values be if you... call the function with *two positional arguments*

```
teach_class(40, "CS2100")
```

num_students=40, name="CS2100", time="2 PM"

What will the parameter values be if you... call the function with *no arguments*

```
teach_class()
```

ERROR!: missing 1 <u>required</u> positional argument

Activity - Use your own function :)

required parameter

```
def teach_class(num_students, name="CS1112", time="2 PM"):
```

What will the parameter values be if you... call the function with *only one positional argument*

```
teach_class(30)
```

num_students=30, name="CS1112", time="2 PM"

What will the parameter values be if you... call the function with *two positional arguments*

```
teach_class(40, "CS2100")
```

num_students=40, name="CS2100", time="2 PM"

What will the parameter values be if you... call the function with *no arguments*

```
teach_class()
```

ERROR!: missing 1 <u>required</u> positional argument

More questions...!

Activity - Use your own function :)

```
def teach_class(num_students, name="CS1112", time="2 PM"):
```

What will the parameter values be if you... call the function with *only the optional arguments*

```
teach_class(name="CS2120", time="3 PM")
```

What will the parameter values be if you... call the function with *one positional argument and an optional argument*

```
teach_class(100, time="11 AM")
```

You can also call *required parameters as keyword arguments*

```
teach_class(num_students=200)
```

More questions...!

Activity - Use your own function :)

```
def teach_class(num_students, name="CS1112", time="2 PM"):
```

What will the parameter values be if you... call the function with *only the optional arguments*

```
teach_class(name="CS2120", time="3 PM")
```

ERROR!: missing 1 <u>required</u> positional argument

What will the parameter values be if you... call the function with *one positional argument and an optional argument*

```
teach_class(100, time="11 AM")
```

You can also call *required parameters as keyword arguments*

```
teach_class(num_students=200)
```

More questions...!

Activity - Use your own function :)

```
def teach_class(num_students, name="CS1112", time="2 PM"):
```

What will the parameter values be if you... call the function with *only the optional arguments*

```
teach_class(name="CS2120", time="3 PM")
```

ERROR!: missing 1 <u>required</u> positional argument

What will the parameter values be if you... call the function with *one positional argument and an optional argument*

```
teach_class(100, time="11 AM")
```

num_students=100, name="CS1112", time="11 AM"

You can also call *required parameters as keyword arguments*

```
teach_class(num_students=200)
```

More questions...!

Activity - Use your own function :)

```
def teach_class(num_students, name="CS1112", time="2 PM"):
```

What will the parameter values be if you... call the function with *only the optional arguments*

```
teach_class(name="CS2120", time="3 PM")
```

ERROR!: missing 1 <u>required</u> positional argument

What will the parameter values be if you... call the function with *one positional argument and an optional argument*

```
teach_class(100, time="11 AM")
```

num_students=100, name="CS1112", time="11 AM"

You can also call *required parameters as keyword arguments*

```
teach_class(num_students=200)
```

num_students=200, name="CS1112", time="2 PM"

A Few More Examples…

```
def getgrade(name, score):
    """ This function computes a grade given a score"""
    if score > 80:
        grade = 'A'
    elif 80 > score > 70:
        grade = 'B'
    elif 70 > score > 60:
        grade = 'C'
    else:
        grade = 'D'

    return name + " had grade: " + grade
```

- **Positional Arguments**
- *To use positional arguments, the arguments need to be passed in the same order as their respective parameters in the function definition.*
- *To call the getgrade() function using positional arguments:*

  `getgrade("Denise", 78)`

  - *This statement automatically passes "Denise" to the "name" parameter and 78 to the "score" parameter.*
- *This function call is not the same as the call above because this statement passes 78 to "name" and "Denise" to "score." And since the score parameter is supposed to be an **integer**, but a **string** is passed to it, it will raise an error and halt the program!* `getgrade(78, "Denise")`

A Few More Examples…

```
def getgrade(name, score):
    """ This function computes a grade given a score"""
    if score > 80:
        grade = 'A'
    elif 80 > score > 70:
        grade = 'B'
```

- **Keyword Arguments**
- *Keyword arguments are arguments that are passed to a function using the name of the argument followed by an equal sign and the value of the argument. These arguments do not need to be passed in a specific order, because the function or method will use the names of the arguments to determine which values to use for which parameters. i.e., passing each argument in the form name = value.*
- *To call the getgrade() function using keyword arguments:*

```
getgrade(name="Denise", score=78)
```

  - *It is very clear that we are assigning "Denise" to "name" and 78 to "score".*
- *You can mix the order in which you provide the arguments:*

```
getgrade(score=78, name="Denise")
```

(Unlike positional arguments, keyword arguments can appear in any order!)

A Few More Examples…

- **Mixing Positional and Keyword arguments**
- *You can mix positional arguments and keyword arguments.*
- *However, the positional arguments cannot appear AFTER any keyword arguments have been defined.*
- *For example, if you have a function header such as:*

```
def func(p1, p2, p3, p4):
```

  - *(A function named "**func**") that has four parameters)*
  - *You can invoke it by using:* `func(21, p2=43, p3=11, p4=7)`
- *It would be wrong to invoke by:*

```
func(p1=21, 43, 11, 7)
```

because the positional arguments 43, 11, and 7 appears AFTER the keyword argument `p1=21`

Python Demonstration

Let's jump on PyCharm!

positional_and_keyword_arguments.py

more_functions.py *(mostly read on your own)*

newton.py and newtontest.py *(**imports**, see direct example)*

In-Class "lab" Activity!

Activity on Importing

- In pairs or groups up to three work on the following activity.
- **sphere.py & spheretest.py**
- *Understand import and import statements. Practicing calling functions.*

Remember to check-in with a TA before leaving class today!