Functions

University of Wyoming COSC 1010

Adapted from: *Python Crash Course 3rd Ed* By Eric Matthes

Functions

- Now we can talk about writing *functions*
- You've been using functions this whole time
- Functions are specific blocks of code defined to do a specific task
- When you want to perform that task you *call*

Functions

- This prevents you from needing to type the code again and again
- Using functions makes writing code easier to read and write
- Information can be passed to functions
- Functions can display information, process data, return a set of values, whatever you need

Defining a Function

- Functions need to be defined before they can be called
- Function definitions need:
  - The `def` keyword
  - The function name
  - A set of parenthesis, optionally with variables inside
  - A colon

In [ ]:
```
def greet_user():
    """This is a simple greeting function"""
    print("Hello there")

greet_user()
```

```
Hello there
```

Defining a Function

- That example is a simple function
- The first line uses the `def` keyword to tell python you are defining a function
- The *function definition* tells python the name of the function, and any information it may need
  - The function name was `greet_user`
  - The empty parenthesis indicate it needs no information

Defining a Function

- Regardless of if the parenthesis are empty, they need to be present
- The second line is known as a *docstring*, it describes what the function does
- When python generates documentation about a function it uses this string
- Typically these strings are enclosed in triple quotes so they can be written on multiple lines

Defining a Function

- The line `print("Hello there")` is the only line of code within the function
- Notice it is indented as with loops and conditional statement code blocks
- Finally to utilize this function a *function call* is needed
- This tells python to call the function
- To call a function you write its name, followed by any information that may be needed

Passing Information to a Function

- The `greet_user()` function can be modified to greet a user by name
- For the function to do this, something needs to be within the parenthesis in the declaration
  - For our purposes `username`
- By adding `username` the function can accept any value specified
- This also means you need to supply something during a function call

In [ ]:
```
def greet_user(username):
    """This is a simple greeting function"""
    print(f"Hello there, {username}")

greet_user("class")
```

```
Hello there, class
```

Passing Information to a Function

- Entering `greet_user("class")` means the function will be called
  - This time with the string `'class'` passed
- Then the function displays the name passed within its output

Arguments and Parameters

- In the preceding `greet_user()` it requires on value for username
- Once the function is called, with information passed, it prints out the passed string
- The variable `username` in the declaration is an example of a *parameter*
  - A piece of information the function needs

Arguments and Parameters

- The value `'class'` in `greet_user("class")` is an example of an *argument*
  - An *argument* is a piece of information that is passed from a function call to a function
- When the function is called the value the function will work with is placed in the parenthesis
- In this case the argument `'class'` was passed to the function `greet_user()`
- The value was assigned to the parameter `username`
- Within the function it can access the value passed via that variable name

Passing Arguments

- Function definitions can have multiple parameters
- A function call may need multiple arguments as a result
- You can pass arguments to a function in a number of ways
  - Positional Arguments
  - Keyword arguments

Passing Arguments

- Positional arguments need to be in the same order the parameters are written
- Keyword arguments use both a variable name and value
- Or, lists and dictionaries of values

Positional Arguments

- When a function is called Python matches each argument in a function call to a parameter
- The easiest way to do so is based on the order of provided arguments
- Values matched this way are called positional arguments
- Consider a function that displays information about pets
- You may need to supply a species and name

In [ ]:
```
def describe_pet(animal_type, pet_name):
    """Display information about a pet"""
    print(f"I have a {animal_type}")
    print(f"My {animal_type}'s name is {pet_name}")

describe_pet("dog","Apollo")
```

```
I have a dog
My dog's name is Apollo
```

Positional Arguments

- The definition shows that the function needs a type of animal, and its name
- When `describe_pet()` is called that information needs to be provided
  - We supplied **dog** for the type and **Apollo** for the name
- In the function body those two parameters are used to display information about the pet

Multiple Function Calls

- A function can be called as many times as needed
- Describing a second pet just needs a second call to the same function
- You can call these functions at any point, you could do so within a loop even
- Calling a function multiple times is an efficient way to work, and the main purpose of functions

In [ ]:
```
def describe_pet(animal_type, pet_name):
    """Display information about a pet"""
    print(f"I have a {animal_type}")
    print(f"My {animal_type}'s name is {pet_name}")

pets = {"Apollo":"dog","Winston":"cat","Flip":"fish", "Harry":"hamster"}

for pet in pets:
    describe_pet(pets[pet],pet)
```

```
I have a dog
My dog's name is Apollo
I have a cat
My cat's name is Winston
I have a fish
My fish's name is Flip
I have a hamster
My hamster's name is Harry
```

Order Matters in Positional Arguments

- If you do not follow the order of arguments defined, odd results can occur
- Thinking on out `describe_pet()` what if we had displaced `"dog"` and `"Apollo`
  - Having: `describe_pet("Apollo","dog")`
  - Rather than: `describe_pet("dog","Apollo")`
- IF you see weird output, check the order of your arguments

In [ ]:
```
def describe_pet(animal_type, pet_name):
    """Display information about a pet"""
    print(f"I have a {animal_type}")
    print(f"My {animal_type}'s name is {pet_name}")

describe_pet("dog","Apollo")
describe_pet("Apollo","dog")
```

```
I have a dog
My dog's name is Apollo
I have a Apollo
My Apollo's name is dog
```

Keyword Arguments

- A *keyword argument* is a name-value pair that is passed to a function
- With this a name can be directly associated with a value within the argument
- This allows the avoidance of mixing up the arguments and their order

In [ ]:
```
def describe_pet(animal_type, pet_name):
    """Display information about a pet"""
    print(f"I have a {animal_type}")
```

```
    print(f"My {animal_type}'s name is {pet_name}")

describe_pet("dog","Apollo")
describe_pet(pet_name="Apollo",animal_type="dog")
```

```
I have a dog
My dog's name is Apollo
I have a dog
My dog's name is Apollo
```

Keyword Arguments

- The definition of the function doesn't need to change
- Instead in the function call what argument aligns with which parameter is explicitly stated
- So, even though the values are passed in reverse order they are still assigned to the correct parameter
- The order of keyword arguments doesn't matter

Default Values

- When writing a function you can define a *default value* for each parameter
- If an argument for a parameter is provided in the function call, Python uses the argument
- Otherwise it uses the functions default value
- When defining a default value you can exclude the corresponding argument if the default value is fine
- Using default values can simplify function calls

In [ ]:
```
def describe_pet(pet_name, animal_type="dog"):
    """Display information about a pet"""
    print(f"I have a {animal_type}")
    print(f"My {animal_type}'s name is {pet_name}")

describe_pet("Apollo")
describe_pet(pet_name="Apollo")
```

```
I have a dog
My dog's name is Apollo
I have a dog
My dog's name is Apollo
```

Default Values

- In that version of the function the definition included a default value for `animal_type`

- Now when the function is called with no `animal_type` provided it will default to `'dog'`
- The order of the parameters in the function definition had to be changed
- Because the default value makes it unnecessary to specify an animal type, we want the name to appear first
- This is so positional arguments can still be used

Equivalent Function Calls

- Positional and keyword arguments along with default values can all be used together
- This will give several equivalent ways to call a function
- it doesn't really matter how you call a function, as long as the results are correct

In [ ]:
```
def describe_pet(pet_name, animal_type="dog"):
    """Display information about a pet"""
    print(f"I have a {animal_type}")
    print(f"My {animal_type}'s name is {pet_name}")

describe_pet("Apollo")
describe_pet("Apollo","dog")
describe_pet(pet_name="Apollo")
describe_pet(animal_type="dog",pet_name="Apollo")
```

```
I have a dog
My dog's name is Apollo
I have a dog
My dog's name is Apollo
I have a dog
My dog's name is Apollo
I have a dog
My dog's name is Apollo
```

Avoiding Argument Errors

- When you start using functions, you may encounter errors about unmatched arguments
- Unmatched arguments occur when fewer or more arguments than are needed are provided
- Utilizing default values can help prevent this

In [ ]:
```
def describe_pet(pet_name, animal_type):
    """Display information about a pet"""
    print(f"I have a {animal_type}")
    print(f"My {animal_type}'s name is {pet_name}")

describe_pet("Apollo")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_1410/3215276067.py in <module>
      4     print(f"My {animal_type}'s name is {pet_name}")
      5
----> 6 describe_pet("Apollo")

TypeError: describe_pet() missing 1 required positional argument: 'animal_type'
```

Avoiding Argument Errors

- The traceback error is helpful in this case
- It identifies that one positional argument is missing, and which one it is
- Python is helpful in that it reads the function's code for us and tells what the names are for the arguments needed
- If you provide too many arguments you would get a similar error

Return Values

- A function doesn't always have to display its output directly
- Sometimes it can process data and return a value or set of
- The value the function returns is called a return value
- The `return` statement takes a value from inside a function
- It then sends it back to the line that called the function
- Return values allow you to move much of a programs work to functions

In [ ]:
```
def get_formatted_name(first_name, last_name):
    """Return a full name formatted properly"""
    return f"{first_name} {last_name}".title()

name = get_formatted_name("cowboy", "joe")
print(name)
```

```
Cowboy Joe
```

Returning a Simple Value

- The definition of `get_formatted_name()` takes as parameters a first and last name
- The function combines these two names, adds a space and returns them in title casing
- When a function that returns a name is called, aa variable needs to be provide
  - To assign the returned value to
- In this case `name` was provided

Returning a Simple Value

- The beauty of functions is that they can be called over again
- This means if you are storing a lot of first and last names separately they can be combined
- Or whatever the use case may be
- Whenever you have a set of code you need to happen multiple times, use a function
- This is better than rewriting it in each individual place

Making an Argument Optional

- Sometimes you may want an argument to be optional
- This expands the functions use case, accepting extra information only when needed
- For example, the format name function could be expanded to include middle names
- Default values can be used to make a value optional

In [ ]:
```
def get_formatted_name(first_name, middle_name, last_name):
    """Return a full name formatted properly"""
    return f"{first_name} {middle_name} {last_name}".title()

name = get_formatted_name("cowboy", "no middle name needed","joe")
print(name)
```

```
Cowboy No Middle Name Needed Joe
```

Making an Argument Optional

- That iteration only includes a middle name parameter
- It however doesn't make it optional
- Middle names aren't always needed and as written that function would require a middle name
- Again, we can utilize default values to make it work properly

In [ ]:
```
def get_formatted_name(first_name, last_name, middle_name=""):
    """Return a full name formatted properly"""
    if middle_name:
        return f"{first_name} {middle_name} {last_name}".title()
    else:
        return f"{first_name} {last_name}".title()

name = get_formatted_name("cowboy", "joe")
print(name)

name = get_formatted_name("John", "Kennedy", "F")
```

```
print(name)
```

```
Cowboy Joe
John F Kennedy
```

Making an Argument Optional

- In that expanded functionality there are two cases
  - One for when a middle name is supplied
  - One for when one isn't
- This allows the function to be called with just a first and last
- Or also including a middle name

Returning a Dictionary

- A function can return any kind of value needed
- This extends beyond strings to also include more complicated data structures
- For example a dictionary could be build to describe someone

In [ ]:
```
def build_person(first_name, last_name):
    """Return a dictionary describing a person"""
    person = {'first:':first_name, 'last':last_name}
    return person

mascot = build_person("pistol","pete")
print(mascot)
```

```
{'first:': 'pistol', 'last': 'pete'}
```

Returning a Dictionary

- The function `build_person()` takes in a first and last name
- It stores these values in a dictionary
- Then the dictionary representing a person is returned
- Finally the dictionary is printed for verification

Returning a Dictionary

- That function can take in simple text and put it into a more meaningful form
- This can be expanded to include optional parameters
- But, a bit more thought needs to be put in than for the previous one
- You can utilize `None` as a default value

- In conditional tests it will evaluate to false

In [ ]:
```
def build_person(first_name, last_name, age = None):
    """Return a dictionary describing a person"""
    person = {'first:':first_name, 'last':last_name}
    if age:
        person["age"] = age
    return person

mascot = build_person("pistol","pete")
print(mascot)

cj = build_person("cowboy","joe",73)
print(cj)
```

```
{'first:': 'pistol', 'last': 'pete'}
{'first:': 'cowboy', 'last': 'joe', 'age': 73}
```

Using a Function with a While Loop

- Functions can be used with all Python concepts we've discussed thus far
- For example `formatted_name()` could be used in a `while` loop to greet users more formally
- A while loop could be used to prompt users continually
- Don't forget an exit condition!

In [ ]:
```
def get_formatted_name(first_name, last_name, middle_name=""):
    """Return a full name formatted properly"""
    if middle_name:
        return f"{first_name} {middle_name} {last_name}".title()
    else:
        return f"{first_name} {last_name}".title()

while True:
    print("Enter q to exit the loop")
    f_name = input("Please enter a first name: ")
    if f_name == 'q':
        break
    l_name = input("Please enter a last name: " )

    if l_name =='q':
        break
    else:
        print(f"Hello there, {get_formatted_name(f_name,l_name)}")
```

```
Enter q to exit the loop
```

Passing a List

- Often it may be useful to pass a list to a function
- When passing a list to a function, the function gets direct access to the contents of a list
- But functions can be used to make working with a list more efficient

In [ ]:
```
def greet_users(names):
    """Print a simple greeting to users in a list"""
    for name in names:
        print(f"Hello there, {name.title()}")

entities = ["steamboat","cowboy joe","pistol pete"]
greet_users(entities)
```

```
Hello there, Steamboat
Hello there, Cowboy Joe
Hello there, Pistol Pete
```

Modifying a List in a Function

- When a list is passed to a function, the function can modify the list
- Any changes made within the function body are permanent
- This allows you to efficiently work with lists, even when there is a large amount of data

In [ ]:
```
list1 = [
    "apple", "banana", "cherry", "dog", "elephant",
    "football", "grape", "happiness", "internet", "jazz",
    "kiwi", "lemon", "mountain", "notebook", "ocean",
    "penguin", "quartz", "rainbow", "sunset", "tiger"
]

# A second list of 20 random words
list2 = [
    "carrot", "dolphin", "elephant", "fox", "guitar",
    "hamster", "igloo", "jellyfish", "kangaroo", "leopard",
    "moon", "narwhal", "octopus", "panda", "quokka",
    "raccoon", "squirrel", "turtle", "umbrella", "violin"
]

def a_purge(words):
    """Purges words with 'a' in them"""
    purge_words = []
    for word in words:

        if "a" in word.lower():
            purge_words.append(word)
    for word in purge_words:
        words.remove(word)

print(list1)
print(list2)

a_purge(list1)
```

```
a_purge(list2)
print("purged:")
print(list1)
print(list2)
```

```
['apple', 'banana', 'cherry', 'dog', 'elephant', 'football', 'grape', 'happiness', '
internet', 'jazz', 'kiwi', 'lemon', 'mountain', 'notebook', 'ocean', 'penguin', 'qua
rtz', 'rainbow', 'sunset', 'tiger']
['carrot', 'dolphin', 'elephant', 'fox', 'guitar', 'hamster', 'igloo', 'jellyfish',
'kangaroo', 'leopard', 'moon', 'narwhal', 'octopus', 'panda', 'quokka', 'raccoon', '
squirrel', 'turtle', 'umbrella', 'violin']
purged:
['cherry', 'dog', 'internet', 'kiwi', 'lemon', 'notebook', 'penguin', 'sunset', 'tig
er']
['dolphin', 'fox', 'igloo', 'jellyfish', 'moon', 'octopus', 'squirrel', 'turtle', 'v
iolin']
```

Modifying a List in a Function

- That function takes in a list
- It then purges all words with an `"a"` in them
- It modifies the original list provided, as demonstrated by the final prints
- Every function should have one specific job to do
- If you need more functionality, more functions!

Preventing a Function from Modifying a List

- Sometimes you'll want to prevent a function from modifying a list
- In this case you can address the issue by passing a passing a copy of the list
- Now any changes made will not be reflected in the original list
- This leaves the original intact

Preventing a Function from Modifying a List

- This is done similarly to how we made a copy of a list earlier
- It is done by slicing the list and passing that slice
  - `listname[:]`
- The slice notation makes a copy of the list and sends it to the function
- Now the original will remain intact

Passing an Arbitrary Number of Arguments

- Sometimes you may not know how many parameters a function will need
- Python allows you to craft a function that can collect an arbitrary number of arguments
- Consider that you have a function that builds a pizza
- You won't necessarily know how many toppings someone may request
- To do so prepend a `*` to the parameter name

In [ ]:
```
def make_pizza(*toppings):
    """Print the list of pizza toppings"""
    print(type(toppings),toppings)

make_pizza("cheese")
make_pizza("cheese","pepperoni","sausage")
```

```
<class 'tuple'> ('cheese',)
<class 'tuple'> ('cheese', 'pepperoni', 'sausage')
```

Passing an Arbitrary Number of Arguments

- The asterisk in the parameter name `*toppings` tell Python to make a `tuple` called `toppings`
- It will contain all the values the function receives
- The print function in the body produces an output showing the tuple
- It will be a tuple even if only one parameter is received

In [ ]:
```
def make_pizza(*toppings):
    """Print the list of pizza toppings"""
    for topping in toppings:
        print(topping)

make_pizza("cheese","pepperoni","sausage")
```

```
cheese
pepperoni
sausage
```

Mixing Positional and Arbitrary Arguments

- If you want a function to accept several kinds of arguments you can!
- The parameter that accepts an arbitrary number of arguments must be placed last in the function definition
- Python matches positional and keyword arguments first and then collects any remaining arguments in the final parameter
- Suppose the pizza function needs to be expanded to include size

In [ ]:
```
def make_pizza(size,*toppings):
```

```
    """Print the list of pizza toppings"""
    print(f"You ordered a {size}\" pizza with the following toppings:")
    for topping in toppings:
        print(topping)

make_pizza(16,"cheese","pepperoni","sausage")
```

```
You ordered a 16" pizza with the following toppings:
cheese
pepperoni
sausage
```

Mixing Positional and Arbitrary Arguments

- In the definition Python assigns the first value received to `size`
- All other values that come after are stored int he tuple
- The function call includes size first
- Then as many toppings as are required can be passed

Using Arbitrary Keyword Arguments

- Sometimes you'll want to accept an arbitrary number of arguments
- But, you may not know ahead of time what information will be passed
- In this case a function can be written that accepts as many key:value pairs as are provided
- Suppose you're building suer profiles, you know you'll get use information but aren't sure what

In [ ]:
```
def build_profile(first, last, **user_info):
    """BUild a dictionary based on information passed"""
    user_info["first_name"] = first
    user_info["last_name"]  = last
    return user_info
user_profile = build_profile("pistol","pete",favcolor="gold",favteam="wyoming")

print(user_profile)
```

```
{'favcolor': 'gold', 'favteam': 'wyoming', 'first_name': 'pistol', 'last_name': 'pet
e'}
```

Using Arbitrary Keyword Arguments

- The definition expects a first and last name, then some number of key:value pairs
- The double asterisk before the parameter causes Python to create a dictionary
  - it will name it `user_info` based on the parameter name

- It will contain all the extra key:value pairs the function receives
- They key:value pairs can be accessed within the function body

Using Arbitrary Keyword Arguments

- You can mix positional, keyword, and arbitrary values in many different ways
- When writing your own functions you have all the power in how they are defined
  - Or called
- It is useful to know that all these argument types exist as they are useful
  - And often used
- it takes practice to use the different types correctly

Styling Functions

- You need to keep a few details in mind when you're styling functions
- Functions should have descriptive names
- Functions should sue lowercase and underscores
- Descriptive names help you (and others) know what the code is trying to do

Styling Functions

- Functions should have a comment that explains concisely what it does
- The comment should appear immediately after the function definition
- It should use the docstring format
- In a well-documented function, other programmers can use it by reading only the description
- They should be able to trust that the code works as described

Styling Functions

- As long as other programmers know the name of the function, the arguments needed, and the return type they should be able to use it in their programs
- If you specify a default value, no spaces should be used on either side of the equals sign
- The same convention should be followed for keyword arguments
- it is recommended that lines are limited to 79 characters
- If your code has more than one function you can separate each by two blank lines