Notes - Tues 01/12/21

Tuesday 01/12/21

Methods and Python Documentation

Objects in Python have methods. So for example the `.split()` and `.append()` are types of methods. A list of the methods can be found by just typing in `mylist.` and seeing the popup window in VSCode. You can also use the help function which would be `help(mylist.mymethod)` which will output help. It's like `man` in the terminal.

For more in-depth documentation, you can go to the official Python documentation.

Intro to Functions

Functions allow us to create blocks of code that can be repeated or executed multiple times. This unlocks more capabilities and problem-solving.

def Keyword - Define a Function

Creating functions uses specific syntax. We use the `def` keyword to define a function. This includes the def keyword, indentation, and proper structure. So let's look at the basic syntax:

```
def name_of_function():
    ```
    Docstring
    ```
    then_your_code_goes_here
```

1. def = Tells Python you're creating a function
2. name_of_function = This is the name of your function. Note it uses "snake casing" which is the lower case words seperated with underscores _. This is the standard Python way to

note functions so it's easy to ID later.
3. parenthesis () = We can pass in arguements or parameters into the function. This sets up future potential.
4. colon : = Indicates to Python and upcoming indented block which will be what's inside the function.
5. Docstring ``` = It's a multiple format to put in comments about the function. I've been using it in Markdown to show lines of code and been using multiple lines of # to do the same thing.
6. Then you write your code that you want the function to execure.

- Note that functions can accept arguements to be passed by the user.

Typically we use `return` keyword to see the result of the function instead of printing it out. We can use `return` to assign the output of the function to a new variable. So this "saves" the output. Return will be covered in depth later.

```
# random method to show what they are
mylist = [1,2,3,4]
mylist.pop()
4
```

```
# help function for methods
help(mylist.pop)


Help on built-in function pop:


pop(index=-1, /) method of builtins.list instance
    Remove and return item at index (default last).


    Raises IndexError if list is empty or index is out of range.
```

Intro to Functions

Functions allow us to create blocks of code that can be repeated or executed multiple times. This unlocks more capabilities and problem-solving.

def Keyword - Define a Function

Creating functions uses specific syntax. We use the `def` keyword to define a function. This includes the def keyword, indentation, and proper structure. So let's look at the basic syntax:

```
def name_of_function():
    ```
    Docstring
    ```
    then_your_code_goes_here
```

1. def = Tells Python you're creating a function
2. name_of_function = This is the name of your function. Note it uses "snake casing" which is the lower case words seperated with underscores _. This is the standard Python way to note functions so it's easy to ID later.
3. parenthesis () = We can pass in arguements or parameters into the function. This sets up future potential.
4. colon : = Indicates to Python and upcoming indented block which will be what's inside the function.
5. Docstring ``` = It's a multiple format to put in comments about the function. I've been using it in Markdown to show lines of code and been using multiple lines of # to do the same thing.
6. Then you write your code that you want the function to execure.

- Note that functions can accept arguements to be passed by the user.

Typically we use `return` keyword to see the result of the function instead of printing it out. We can use `return` to assign the output of the function to a new variable. So this "saves" the output. Return will be covered in depth later.

```
# Let's create a function where we're adding numbers


def add_function(num1,num2):
    return num1+num2


# Now with return, we're able to save the output of the function as the "result"
# We want add_function to run with the numbers 1 and 2
result = add_function(1,2)


# Now we can see what the result from above
print(result)
3
```

Basic Functions in Python Practice

Let's create a function called `say_hello` which will output Hello and then a name. In this example I define the function as ```say_hello``. Then on the next indented line, I tell python what the function does. Then run the function to get the output.

```
def say_hello():
    print('smell')
    print('you later')

say_hello()

smell
you later
```

Function Variable

Now let's add in a parameter variable which we can pass using an f-string literal. So we'd add 'name into the parathesis after `say_hello` and then the next line has an f-string literal which contains the variable. Then we want to output the function but add in a name. Note that the variable 'name' could be anything but we want it to be easily understood. For example I used this to show how many tries were left in my Temperature Converter Program.

Basic Syntax would be `def ur_function(ur_variable)`

```
def say_hello(name):
    print(f'Hello {name}')

say_hello('Girlshark')

Hello Girlshark
```

Note that if I run `say_hello()`, I'll get a "positional arguement error" which means I missed putting something in for the name.

```
say_hello()



TypeError                                 Traceback (most recent call last)
<ipython-input-32-faa5fc24272a> in <module>
----> 1 say_hello()

TypeError: say_hello() missing 1 required positional argument: 'name'
```

Default Value

Let's say that we don't want that error to happen. We can setup something called a "default value". So this is the value that would show up if we missed inputting something after `say_hello`. But then it'd work normally after we put in an input.

Basic Syntax would be `def ur_function(ur_variable='default_value')`

```
def say_hello(name='stupid; You forgot to put something in'):
    print(f'Hello {name}')


say_hello()


Hello stupid, You forgot to put something in
```

```
say_hello('wizbro')


Hello wizbro
```

Return Keyword

As mentioned above, we typically don't have functions just print stuff out. Otherwise we'd just print them out instead of setting up a function. Instead, we want to use the `return` keyword. `print` just shows the output while `return` allows you to save the output as a variable.

Note in the below example, we've set 2 variables in the define function (aka first line) line.

Basic Syntax would be (note there's no parenthesis when using return):

```
def ur_function(ur_variables):
    return do_something_to_variables
```

```
def add_num(num1,num2):
    return num1+num2


# See I saved it
result = add_num(10,20)
result


30
```

Let's compare what happens when we try to save the output from `print` vs `return`. Note that we can use these together. It just yields a different output.

```
# Let's compare print vs return
def print_result(a,b):
    print(a+b)

def return_result(a,b):
    return a+b

# I tried to save the result of print_result then check the type.
# Note that the result is nonetype

result = print_result(10,20)
type(result)

30
NoneType
```

```
# Now compare that the the return function
# I saved the return result then output it

result2 = return_result(10,20)
result2

30
```

```
# Then I checked the type which gives me integer
type(result2)

int
```

```
# Combine print + result
# This isn't 100% needed. Good for things like troubleshooting.

def myfunc(a,b):
    print(a+b)
    return a+b

result = myfunc(665,1)
```

```
666
```

Note that python is dynamically typed so you don't have to tell Python what data/object type you're inputting before you run the code. So let's say you've got user input. That `input` value yields a string and you might want a number instead.

Functions with Logic

Reminder that logic is stuff like if or else statements.

For example, let's write a function that checks if a number is even.

```
def even_check(number):
    result = number % 2 == 0
    return result

even_check(20)

True

even_check(21)

False
```

Above has the noob version of the definition. But you can compress the return line if there's a boolean check.

So the above result is: `result = number % 2 == 0`

And the advanced mode is: `return number % 2 == 0`

```
def even_check(number):
    return number % 2 == 0

even_check(16)

True
```

Now let's check a list. We want to return True if any number is even inside the list.

```
def check_even_list(num_list):
    for number in num_list:
    # check all the variable objects in the defined list
        if number % 2 == 0:
        # if any numbers have a remainder of 0 after dividing by 2
            return True
        # return the True boolean
        else:
            pass
        # otherwise pass or end the code

check_even_list([1,2,3])


True
```

```
check_even_list([3,5,7])
# If I run this list, I currently only output if there's something even. So there's no output. That's what the pass does.


def check_even_list(num_list):
    for number in num_list:
    # check all the variable objects in the defined list
        if number % 2 == 0:
        # if any numbers have a remainder of 0 after dividing by 2
            return True
        # return the True boolean
        else:
            pass
        # otherwise pass or end the code
```

Now let's have an output that when there's a non-even number, we get False. If we just modify `pass` in the last night to `return False`, the code will only run once since we're essentially telling the code to return True and False at the same time. Instead, we need to setup the code so that there's multiple loops. First is the even number = output True loop. Then we setup a second loop to check for odd numbers. So the new return False line would align with the For loop.

```
def check_even_list(num_list):
    for number in num_list:
        if number % 2 == 0:
            return True
```

```
        else:
            pass
# Above the code is what we had before.

    return False
# So now what this is saying, run the first loop to check even number = True.
# If that loop doesn't work, the code is stopped with pass and
# then it will return False

check_even_list([1,2,4])

True

check_even_list([1,3,5])

False
```

Ok now let's say we want the program to return all the even numbers in a list. Otherwise it will return an empty value. Note you can setup a placeholder function for variables you're not sure of yet.

```
def check_even_list(num_list):
    even_numbers = []
    # Here is the empty placeholder list

    for number in num_list:
        if number % 2 == 0:
            even_numbers.append(number)
            # Here we're editing the even_number variable using append
            # Remember append adds new objects to the end of the list
            # so we're setting a variable and checking that variable to be even
            # then we're adding those numbers to the list

        else:
            pass
            # if this doesn't work out, it'll just stop the program

    return even_numbers
    # here we're returning the even_numbers variable
```

```
check_even_list(range(0,11))
# finally we're running the function with the range 0-10


[0, 2, 4, 6, 8, 10]


check_even_list([1,3,5,7])
# running another list with no even numbers and we get an empty list


[]
```

Functions and Tuple Unpacking

Now we're going to return mulitiple items using tuple unpacking which was discussed previously. Let's run an example to remind me how Tuple Unpacking works.

```
stock_prices = [('APPL',200),('GOOG',400),('MSFT',100)]


for item in stock_prices:
    print(item)


('APPL', 200)
('GOOG', 400)
('MSFT', 100)


for ticker,price in stock_prices:
    print(price-(.15*price))
    # we're unpacking the price but also
    # showing the price with a 15% loss - F


170.0
340.0
85.0
```

Turns out we can use tuple unpacking for a function too. Let's setup a new example where we have a list of work hours and we want to find the employee of the year.

```
work_hours = [("Krabs",32),("Squidward",40),("Spongebob",2000)]
# Here's a list of the Krusty Krab employees and their hours
```

```
def employee_check(work_hours):
# we're setting up the function and variable

    current_max = 0
    employee_of_month = ""
    # This sets the current max variable as a placeholder for the hours later
    # and the employee of the month variable as a placeholder

    for employee,hours in work_hours:
        if hours > current_max:
        # this checks the hours from the tuple
        # if it's more the current_max hours
        # it resets the variables current_max and employee_of_the_month
            current_max = hours
            employee_of_month = employee
        else:
            pass
        # if the above loop is finished, then the code stops

    return (employee_of_month,current_max)
    # this returns the variables

result = employee_check(work_hours)
result
# this stores the function output as a new variable
# the current employee_check is now Spongebob

('Spongebob', 2000)

# We can do tuple unpacking on the function now
name,hours = employee_check(work_hours)
hours

2000

result

('Spongebob',2000)
```

Interactions Between Functions

Typically a Python script, code, program will contain several functions interacting with each other. We'll take the results of one function and use them as inputs to another function. To show this, we're going to write a 3 cup monte shuffle game. We won't see the graphics but mimic the effect with lists. And we're not going to see the shuffle. The user will randomly guess.

First, let's have a reminder that the `random` library exists. That library has a `shuffle` function which will randomy shuffle a list of numbers. So let's look at that.

Note that a function will be needed to store the `shuffle` output.

```
from random import shuffle
# this imports shuffle from the random library

example_list = [1,2,3,4,5,6,7]
# this generates a list from 0-7 called example_list

shuffle(example_list)
# note this can't be stored and can't be returned as is

example_list

[1, 3, 6, 4, 7, 5, 2]

def shuffle_list(mylist):
    shuffle(mylist)
    return mylist
# this function stores the shuffle so I can use it repeatedly

result = shuffle_list(example_list)
# variable result uses function shuffle_list to shuffle the example_list i had above
result

[2, 3, 1, 4, 5, 7, 6]

# Now we can simulate O as the ball in the cup shuffle game
# let's setup a list then use the shuffle_list function from above
mylist = ['','O','']
```

```
shuffle_list(mylist)
```

```
['O', '', '']
```

```
# Now let's setup the player's guess
def player_guess():

    guess=''

    while guess not in ['0','1','2']:
        guess = input("Pick a location: 0, 1, 2")

    return int(guess)

player_guess()
```

```
2
```

```
# Now let's write a function to check the guess

def check_guess(mylist,guess):
    if mylist[guess] == "O":
        print("Correct!")

    else:
        print("Wrong guess!")
        print(mylist)

# Now let's combine it all

# Initial List
mylist = ['','O','']

# Shuffle List
mixedup_list = shuffle_list(mylist)

# User Guess
guess = player_guess()
```

```
# Check Guess
check_guess(mixedup_list,guess)


Wrong guess!
['O', '', '']
```

Overview of Quick Function Exercises #1-10 (aka notes before the Quiz coming up)

Okay so you now know functions!

A big part of this section of the course will be testing your new skills with exercises. We have 3 main parts of exercises.

Part 1: 10 In Course Coding Exercises

We're going to start off with just the basics with a series of 10 problems. These problems should feel relatively easy, just some quick exercises to get you comfortable with the syntax of functions. If you feel uncomfortable with these, check out lecture 26 for some useful links for warm-up problems from codingbat.com , but hopefully these exercises should feel relatively easy.

These are in-course coding exercises. Solutions can be found linked here:

https://docs.google.com/document/d/181AMuP-V5VnSorl_q7p6BYd8mwXWBnsZY_sSPA8trfc/edit?usp=sharing

In between these in-course coding exercises we'll have a quick lecture on *args and **kwargs.

Part 2: Function Practice Exercises

Here we'll have a jupyter notebook with some exercises for you to answer, we'll have a quick overview lecture, and then have you attempt problems, afterwards we'll have an explanatory solutions video. These problems are ranked WarmUp, Level 1, Level 2, and Challenge. You should feel comfortable with Warmup and Level 1 and Level 2. Challenge problems here are very difficult, so don't feel bad if you don't want to attempt them yet! :)

After this we'll cover a few more topics through some videos.

Part 3: Function and Methods Homework

We finish off this section with even more exercises! Here we have various function word problems for you to solve, again in a notebook and we will cover the solutions in a video afterwards.

Best of luck! If you have any questions, post to the QA forums and we'll be happy to help you out!

Revision #2
Created 12 January 2021 23:33:46 by cba88
Updated 13 January 2021 07:35:29 by cba88