Class Notes For Python

A step-by-step guide to level up your Knowledge in Python

Print a message onto the screen:

The `print()` function prints the specified message to the screen, or other standard output device.

The message can be a string, or any other object, the object will be converted into a string before being written to the screen.

```
print("Hello world!")
```
Hello world!

```
print('Hello world!')
```
Hello world!

Single and duble quotes do the same job but in case you are printing a string with upstroph you should use the duble quotes like the code below

```
print("let's start!.")
```

If you want to print a long string (in multiple lines) you should use the triple quotes

```
print('''
Hello world!
My name is ali
I am a python developer''')
```
Output
Hello world!
My name is ali
I am a python developer

Variables and Simple Data Types

Variables are containers for storing data values, Python has no command for declaring a variable, a variable is created the moment you first assign a value to it.

```
x = 5
y = "John"
print(x)
print(y)
```

Variables do not need to be declared with any particular *type*, and can even change type after they have been set.

```
x = 4       # x is of type int
x = "Sally" # x is now of type str
print(x)
```

Variable Names

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume).

Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Legal variable names:

```
myvar = "Ahmed"
my_var = "Ahmed"
_my_var = "Ahmed"
myVar = "Ahmed"
MYVAR = "Ahmed"
myvar2 = "Ahmed"
```

Illegal variable names:

```
2myvar = "Ahmed"
my-var = "Ahmed"
my var = "Ahmed"
```

Avoiding Name Errors When Using Variables

You should write the variable name without any mistakes

```
message = "Hello Python"
print(messag)
```

If you forget the “e” letter this error will rise NameError: name 'messag' is not defined

Data Types

Strings

A string is a series of characters. Anything inside quotes is considered a string in Python, and you can use single or double quotes around your strings

```
name = "Ali"
```

Changing Case in a String with Methods

`print(name.upper())` this prints >> `ALI`

`print(name.lower())` this prints >> `ali`

You can print the variables values with print

```
first_name = "Mahmoud"
last_name = "Ali"
print(first_name, last_name )
```

Output

```
Mahmoud Ali
```

To add a tab to your text, use the character combination `\t` as shown

```
print("\tPython")
```

Output

```
    Python
```

`\n` means new line

```
print("Languages:\nPython\nC\nJavaScript")
```

Output

Languages:
Python
C
JavaScript

String concatenation

```
print('mahmoud'+ 'hassan')
```

Output

mahmoudhassan

Fstring

```
first_name = "mahmoud"
last_name = "hassan"
full_name = f"{first_name} {last_name}"
print(full_name)
```

Output
mahmoud hassan

Numbers

Integers

```
Number = 100
```

You can use some basic mathematical operations with `print()`

```
print(2 + 3)
5

print(2 * 3)
6

print(2 ** 3)
8

print(2 / 3)
0.66666


print((1+2)*4)
12
```

Try It Yourself

Write addition, subtraction, multiplication, and division operations that each result in the number 8. Be sure to enclose your operations in print() calls to see the results. You should create four lines that look like this: print(5+3)

Floats

Python calls any number with a decimal point a float

```
print(0.1 + 0.5)
0.6
```

_____

When you divide any two numbers, even if they are integers that result in a whole number, you'll **always get a float:**

4/2 ⇒ 2.0

_____

1 + 2.0 ⇒ 3.0

Multiple Assignment

You can assign values to more than one variable using just a single line. This can help shorten your programs and make them easier to read; you'll use

this technique most often when initializing a set of numbers. For example, here's how you can initialize the variables x, y, and z

```
x, y, z = 0, 1, 2
```

Constants

A constant is like a variable whose value stays the same throughout the life of a program. Python doesn't have built-in constant types, but Python programmers use all capital letters to indicate a variable should be treated as a constant and never be changed:

```
MAX_CONNECTIONS = 5000
```

When you want to treat a variable as a constant in your code, make the name of the variable all capital letters.

Comments

As your programs become longer and more complicated, you should add notes within your programs that describe your overall approach to the problem you're solving. A comment allows you to write notes in English within your programs.

```
# Say hello to everyone.
print("Hello Python people!")
```

You can write long comments by using triple quotes

```
'''
My name is mahmoud ali
Age : 14
Address : Egypt
'''
```

List

You can make a list that includes the letters of the alphabet, the digits from 0–9, or the names of all the people in your family. You can put anything you

want into a list, the items in your list don't have to be related in any particular way. Because a list usually contains more than one element, it's a good idea to make the name of your list plural, such as letters, digits, or names.

```
names = ['mahmoud', 'ali', 'ahmed', 'mohamed']
```

```
my_list = ['mahmoud',20, 'ahmed', '300']
```

Accessing Elements in a List

NOTE: Index Positions Start at 0, Not 1

```
names = ['mahmoud', 'ali', 'ahmed', 'mohamed']
print(names[0])
```

Output

```
'mahmoud'
```

the last element in your list will have the index of -1

```
print(names[-1])
```

Output

```
'mohamed'
```

Changing, Adding, and Removing Elements

Modifying Elements in a List

```
names = ['mahmoud', 'ali', 'ahmed', 'mohamed']
print(names)
```

Output

```
['mahmoud', 'ali', 'ahmed', 'mohamed']
```

```
names[0] = 'Max'
print(names)
```

Output

['Max', 'ali', 'ahmed', 'mohamed']

Adding Elements to a List

```
names.append('Basma')
```

You can append (add) any items to your empty list like:

```
names = []
names.append('Basma')
names.append('Ali')
names.append('samer')
print(names)
```

Output

['Basma', 'Ali', 'samer']

extend takes a list as an argument and **appends all** of the elements:

```
t1 = ['a', 'b', 'c']
t2 = ['d', 'e']
t1.extend(t2)
print( t1)
```

Output

['a', 'b', 'c', 'd', 'e']

Inserting Elements into a List

You can add a new element at any position in your list by using the insert() method. You do this by specifying the index of the new element and the value of the new item.

```
motorcycles = ['honda', 'yamaha', 'suzuki']
motorcycles.insert(0, 'ducati')
print(motorcycles)
```

Output

['ducati', 'honda', 'yamaha', 'suzuki']

Removing an Item Using the *del* Statement
If you know the position of the item you want to remove from a list, you can use the **del** statement.

```
motorcycles = ['honda', 'yamaha', 'suzuki']
del motorcycles[0]
print(motorcycles)
```
Output
['yamaha', 'suzuki']

Removing an Item Using the *pop()* Method

Sometimes you'll want to use the value of an item after you remove it from a list. For example, you might want to get the x and y position of an alien that was just shot down, so you can draw an explosion at that position.

In a web application, you might want to remove a user from a list of active members and then add that user to a list of inactive members.

The pop() method removes the last item in a list, but it lets you work with that item after removing it. The term pop comes from thinking of a list as a stack of items and popping one item off the top of the stack. In this analogy, the top of a stack corresponds to the end of a list.
Let's pop a motorcycle from the list of motorcycles:

```
popped_name = names.pop()
print(popped_name)

motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

popped_motorcycle = motorcycles.pop()
print(motorcycles)
print(popped_motorcycle)
```

Output
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha']
suzuki

Popping Items from any Position in a List You can use *pop()* to remove an item from any position in a list by including the index of the item you want to remove in parentheses.

```
motorcycles = ['honda', 'yamaha', 'suzuki']
popped motorcycle = motorcycle.pop(0)
```

Output
honda

Removing an Item by Value

Sometimes you won't know the position of the value you want to remove from a list.

```
names = ['mahmoud', 'ali', 'ahmed', 'mohamed']
names.remove('ali')
```

The remove() method deletes only the first occurrence of the value you specify. If there's a possibility the value appears more than once in the list, you'll need to use a loop to make sure all occurrences of the value are removed.

Organizing a List

Your lists will be created in an unpredictable order, because you can't always control the order in which your users provide their data. Although this is unavoidable in most circumstances, you'll frequently want to present your information in a particular order. Sometimes you'll want to preserve the original order of your list, and other times you'll want to change the original order. Python provides a number of different ways to organize your lists, depending on the situation.

Sorting a List

```
numbers = [5, 3, 1, 4,2]
numbers.sort()
numbers.sort()
print(numbers)
```

Output
[1, 2, 3, 4, 5]

```
numbers.reverse()
```

Output
[5, 4, 3, 2, 1]

Notice that reverse() doesn't sort backward.

Sorting a list of strings

```
names = ['mahmoud', 'ali', 'ahmed', 'mohamed']
names.sort()
print(names)
```

Output
['ahmed', 'ali', 'mahmoud', 'mohamed']

Finding the Length of a List

You can quickly find the length of a list by using the *len()* function. The list in this example has four items, so its length is 4:

```
names = ['mahmoud', 'ali', 'ahmed', 'mohamed']
print(len(names))
```
4

Avoiding Index Errors When Working with Lists

Let's say you have a list with three items, and you ask for the fourth item:

```
names = ['mahmoud', 'ali', 'ahmed', 'mohamed']
print(names[4])
```
Output
IndexError: list index out of range

Python attempts to give you the item at index 3. But when it searches the list, no item in names has an index of 3.
Because of the off-by-one nature of indexing in lists, this error is typical. People think the third item is item number 3, because they start counting at 1. But in Python the third item is number 2, because it starts indexing at 0.

An index error means Python can't find an item at the index you requested. If an index error occurs in your program, try adjusting the index you're asking for by one. Then run the program again to see if the results are correct.

Keep in mind that whenever you want to access the last item in a list you use the index **-1**
This will always work, even if your list has changed size since the last time you accessed it

Looping Through an Entire List

Let's say we have a list of names, and we want to print out each name in the list

```
names = ['mahmoud', 'ali', 'ahmed', 'mohamed']
for name in names:
    print(name)
```

Output

mahmoud
ali
ahmed
mohamed

Doing Something After a for Loop

What happens once a for loop has finished executing? Usually, you'll want to summarize a block of output or move on to other work that your program must accomplish. Any lines of code after the for loop that are not indented are executed once **without repetition**.
we place the end message after the for loop without indentation:

```
names = ['mahmoud', 'ali', 'ahmed', 'mohamed']
for name in names:
    print(name)
print('the end')
```

Output

mahmoud
ali
ahmed
mohamed
the end

Making Numerical Lists

Many reasons exist to store a set of numbers. For example, you'll need to keep track of the positions of each character in a game, and you might want to keep track of a player's high scores as well.

In data visualizations, you'll almost always work with sets of numbers, such as temperatures, distances, population sizes, or latitude and longitude values, among other types of numerical sets. Lists are ideal for storing sets of numbers, and Python provides a variety of tools to help you work efficiently with lists of numbers.

Once you understand how to use these tools effectively, your code will work well even when your lists contain millions of items.

```
numbers = list(range(1, 6))
print(numbers)
```

Output

[1, 2, 3, 4, 5]

```
even_numbers = list(range(2, 11, 2))
print(even_numbers)
```

In this example, the range() function starts with the value 2 and then adds 2 to that value. It adds 2 repeatedly until it reaches or passes the end value, 11, and produces this result:

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

```
squares = []
for value in range(1, 11):
    square = value ** 2
    squares.append(square)

print(squares)
```

Simple **Statistics** with a List of Numbers

```
digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]

print(min(digits))
0

print(max(digits))
9

print(sum(digits))
45
```

List Comprehensions

```
squares = [value**2 for value in range(1, 11)]
print(squares)
```

Output

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Try It Yourself

3.1 **Guest List**: If you could invite anyone, living or deceased, to dinner, who would you invite?

Make a list that includes at least three people you'd like to invite to dinner. Then use your list to print a message to each person, inviting them to dinner.

3.2 **Changing Guest List**: You just heard that one of your guests can't make the dinner, so you need to send out a new set of invitations. You'll have to think of someone else to invite.

- Start with your program from Exercise 3.1. Add a print() call at the end of your program stating the name of the guest who can't make it.
- Modify your list, replacing the name of the guest who can't make it with the name of the new person you are inviting.
- Print a second set of invitation messages, one for each person who is still in your list.

3.3 **More Guests**: You just found a bigger dinner table, so now more space is available. Think of three more guests to invite to dinner.

- Start with your program from Exercise 3.1 or Exercise 3.2. Add a print() call to the end of your program informing people that you found a bigger dinner table.
- Use *insert()* to add one new guest to the beginning of your list.
- Use *insert()* to add one new guest to the middle of your list.
- Use *append()* to add one new guest to the end of your list.
- Print a new set of invitation messages, one for each person in your list

Slicing a List

Working with Part of a List

```
names = ['mahmoud', 'ali', 'ahmed', 'mohamed']
print(names[0:3])
print(names[0:4])
```

Output

['mahmoud', 'ali', 'ahmed']
['mahmoud', 'ali', 'ahmed', 'mohamed']

```
print(names[:4])
print(names[2:])
print(names[-3:])
```

Output

['mahmoud', 'ali', 'ahmed', 'mohamed']
['ahmed', 'mohamed']
['ali', 'ahmed', 'mohamed']

Looping Through a Slice

```
for name in names[:3]:
    print(name)
```

Output

mahmoud
ali
ahmed

Copying a List

```
my_foods = ['pizza', 'falafel', 'cake']
friend_foods = my_foods[:]
print(my_foods)
print(friend_foods)
```

Output
['pizza', 'falafel', 'cake']
['pizza', 'falafel', 'cake']

Tuples

Python refers to values that cannot change as ***immutable***, and **an immutable list is called a tuple.**

```
dimensions = (200, 50)
print(dimensions[0])
```

Output
```
(200, 50)
```

You can not modify the values in tuples
Error
dimensions[0] = 250

Dictionaries

A dictionary in Python is a collection of **key-value pairs.** Each **key** is connected to a **value,** and you can use a **key to access the value associated with that key.**
A key's value can be a number, a string, a list, or even another dictionary.

```
user = {'Name': 'Ali', 'Age': 25}
print(user ['Name'])
print(user ['Age'])
```

Output
Ali
25

Adding New Key-Value Pairs

```
user = {'Name': 'Ali', 'Age': 25}
user['Address'] = 'Egypt'
user['weight'] =  70
print(user)
```

Output
{'Name': 'Ali', 'Age': 25, 'Address': 'Egypt', 'weight': 70}

Modifying Values in a Dictionary

```
user['Address']= 'USA'
```

Removing Key-Value Pairs

```
del user['Address']
```

A Dictionary of Similar Objects

```
favorite_languages = {
'mohamed': 'python',
'sara': 'c',
'emad': 'ruby',
'mona': 'python',
}
```

if the key you ask for doesn't exist, you'll get an error.
print(favorite_languages['ola'])
Using get() **to Access Values**
ola_value = favorite_languages.**get('ola', 'No ola value assigned.')**

print(ola_value)

Looping Through a Dictionary

```
for key, value in user.items():
    print(f"\nKey: {key}")
    print(f"Value: {value}")
```

```
for name in favorite_languages.keys():
   print(name)
```

```
for name in favorite_languages.values():
    print(name)
```

---

if Statements

Simple if Statements

if **conditional_test :**

do something

```
age = 19
print(age==17)
False
```

A single equal sign is really a statement; you might read the code as “**Set the value of age equal to 19**.”

On the other hand, a double equal sign (==) , asks a question: “**Is the value of age equal to 17?**”

```
age = 19
if age == 19:
   print("You can vote!")
```

!=

```
age = 19
if age != 15:
  print("You can not vote!")
```

Checking Whether a Value Is in a List

```
names = ['Ali', 'mohamed', 'mona']
print('mona' in names)
print('ahmed' in names)
```

Checking Whether a Value Is Not in a List

```
banned_users = ['Ali', 'mohamed', 'mona']
user = 'max'
if user not in banned_users:
   print(f"{user}, is not in the banned users list")
```

---

if-else Statements

```
my_list = ['samy', 'bmw', 'mona', 'toyota']
for i in my_list:
   if i == 'bmw':
       print(i.upper())
   else:
       print(i.title())
```

Boolean Expressions

```
active = True
offline = False
```

The if-elif-else Chain

```
age = 20
if age < 30:
   print("your offer is 30%")
elif age < 50:
   print("your offer is 50%")
else:
   print("your offer is 60%")
```

Using Multiple elif Blocks

```
age = 20
if age < 30:
   print("your offer is 30%")
elif age < 50:
   print("your offer is 50%")
elif age < 60:
   print("your offer is 60%")
else:
   print("your offer is 70%")
```

Note, Python does not require an else block at the end of an if-elif chain.

```
age = 20
if age < 30:
   print("your offer is 30%")
elif age < 50:
   print("your offer is 50%")
elif age < 60:
   print("your offer is 60%")
```

No else statement.

User Input

```
message = input("Tell me something, and I will repeat it back to
you: ")
print(message)
```

```
prompt = "If you tell us who you are, we can personalize the messages
you see."
prompt += "\nWhat is your first name? "
name = input(prompt)
print(f"\nHello, {name}!")
```

```
name = input("Please enter your name: ")
print(f"\nHello, {name}!")
```

When you use the `input()` function, Python interprets everything the user enters as a **string.**

```
age = input("How old are you? ")
print(type(age))
```

Error

```
age = input("How old are you? ")
if age >= 18 :
   print('you can vote')
```

TypeError: '>=' not supported between instances of **'str' and 'int'**

Using int() to Accept Numeric Input

```
age = input("How old are you? ")
age = int(age)
if age >= 18:
   print('you can vote')
```

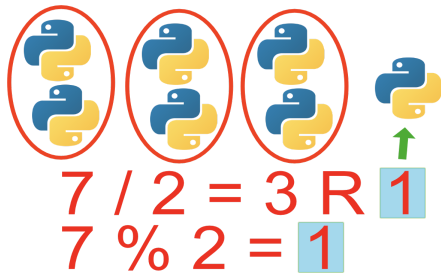
```
score = input("Enter Score Grade:")
score = int(score)
if score < 60:
   print("Your grade is an F")
elif score < 70:
   print("grade d")
elif score < 80:
   print("grade c")
elif score < 90:
   print("grade b" )
elif score <= 100:
   print("grade a")
else:
   print("wrong score")
```

```
print(65 / 3)
```
⇒ **21.666666666666668**

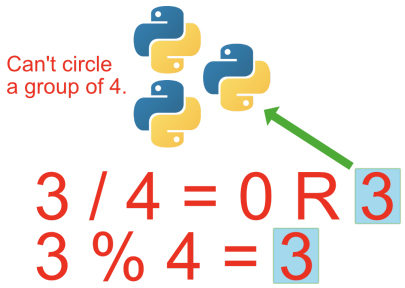
```
print(65 // 3)
```
⇒ **21**

The Modulo Operator
which divides one number by another number and returns the **remainder**

```
print(7 % 2)
```
⇒ 1




```
print(3 % 4)
```
⇒ 0



while Loops

```
number = 1
while number <= 5:
    print(number)
    number += 1
```

Letting the User Choose When to Quit

```
prompt = "\nTell me something, and I will repeat it back to
you:"
prompt += "\nEnter 'quit' to end the program. "

message = ""
while message != 'quit':
    message = input(prompt)
    print(message)
```

Using a Flag

```
prompt = "\nTell me something, and I will repeat it back to
you:"
prompt += "\nEnter 'quit' to end the program. "

active = True
while active:
    message = input(prompt)
    if message == 'quit':
        active = False
    else:
        print(message)
```

Using break to Exit a Loop

```
prompt = "\nPlease enter the name of a city you have visited:"
prompt += "\n(Enter 'quit' when you are finished.) "
while True:
    city = input(prompt)
    if city == 'quit':
        break
    else:
        print(f"I'd love to go to {city.title()}!")
```

Note; You can use the break statement in any of Python's loops. For example, you could use break to quit a for loop that's working through a list or a dictionary.

Using continue in a Loop

```
current_number = 0
while current_number < 10:
    current_number += 1
    if current_number % 2 == 0:
        continue
    print(current_number)
```

Avoiding Infinite Loops

```
# right
x = 1
while x <= 5:
    print(x)
    x += 1

# infinite loop
x = 1
while x <= 5:
    print(x)
```

Defining a Function

```
# function definition.
def say_Hello():
    """Print hello."""
    print("Hello!")

# function call
say_Hello()
```

Passing Information to a Function

```
def say_Hello(username):
    """Display a simple greeting."""
    print(f"Hello, {username.title()}!")
say_Hello('ahmed')
```

Arguments and Parameters

The variable `username` in the definition of `say_Hello()` is an example of a **parameter**, a piece of information the function needs to do its job.
The value `'ahmed'` in `say_Hello('ahmed')` is an example of an **argument**.
An argument is a piece of information that's passed from a function call to a function.

Passing Arguments

Because a function definition can have multiple **parameters**, a function call may need multiple **arguments**. You can pass arguments to your functions in a number of ways. You can use **positional arguments**, which need to be in the same order the parameters were written; **keyword arguments**, where each argument consists of a variable name and a value.

Positional Arguments

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")
describe_pet('dog', 'harry')
```

Multiple Function Calls

```
describe_pet('dog', 'harry')
describe_pet('cat', 'lilly')
describe_pet('dog', 'max')
```

Order Matters in Positional Arguments

```
describe_pet('max','dog')
```

Keyword Arguments

A keyword argument is a name-value pair that you pass to a function. You directly associate the name and the value within the argument, so when you pass the argument to the function, there's no confusion(you won't end up with a max named dog). **Keyword arguments** free you from having to worry about correctly ordering your arguments in the **function call**, and they clarify the role of each value in the function call.

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")
describe_pet(animal_type='dog', pet_name='max')
```

OR

```
describe_pet(animal_type='dog', pet_name='max')
describe_pet(pet_name='cat', animal_type='lilly')
```

Quiz

```
def about_user(user_name, phone,age,address):
    """Display information about a pet."""
    print(f"\nyour name is: {user_name.title()}.")
    print(f"phone : {phone}")
    print(f"Age : {age}")
    print(f"Address : {address.title()}")

about_user(user_name='mahmoud hassan', phone='01024662683', age=20, address='egypt' )
```

Default Values

```
def about_user(user_name, phone,age,address='Egypt'):
    """Display information about a pet."""
    print(f"\nyour name is: {user_name.title()}.")
    print(f"phone : {phone}")
    print(f"Age : {age}")
    print(f"Address : {address.title()}")
about_user(user_name='mahmoud hassan', phone='01024662683', age=20)
```

OR

```
def about_user(user_name, phone='010246626',age=20,address='Egypt'):
    """Display information about a pet."""
    print(f"\nyour name is: {user_name.title()}.")
    print(f"phone : {phone}")
    print(f"Age : {age}")
    print(f"Address : {address.title()}")
about_user('mahmoud')
```

Equivalent Function Calls

```
def describe_pet(pet_name, animal_type='dog'):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

# A dog named max.
describe_pet('max')
describe_pet(pet_name='max')

# A cat named lilly.
describe_pet('lilly', 'cat')
describe_pet(pet_name='lilly', animal_type='cat')
describe_pet(animal_type='cat', pet_name='lilly')
```

Avoiding Argument Errors

describe_pet()

Return Values

```
def get_name(first_name, last_name):
    """Return a full name, neatly formatted."""
    full_name = f"{first_name} {last_name}"
    return full_name.title()

name_from_function = get_name('mahmoud', 'hassan')
print(name_from_function)
```

Making an Argument Optional

```
def get_name(first_name, last_name, middle_name=''):
    """Return a full name, neatly formatted."""
    if middle_name:
        full_name = f"{first_name} {last_name} {middle_name}"
    else:
        full_name = f"{first_name} {last_name}"
    return full_name.title()


name = get_name('mahmoud', 'hassan')
print(name)
name= get_name('mahmoud', 'hassan', 'mohamed')
print(name)
```

Quiz

Passing a List

Print this output by passing a list of names to a function called `say_hello_username()`

⇒ **Hello, Ali!**
Hello, Samy!
Hello, Mahmoud!

solution

```
def say_hello_username(names):
    """Print a simple greeting to each user in the list."""
    for name in names:
        msg = f"Hello, {name.title()}!"
        print(msg)

usernames = ['ali', 'samy', 'mahmoud']
say_hello_username(usernames)
```

Storing Your Functions in Modules

test.py

```
def say_hello_username(names):
    """Print a simple greeting to each user in the list."""
    for name in names:
        msg = f"Hello, {name.title()}!"
        print(msg)
```

Now we'll make a separate file called ***say.py*** in the same directory as ***test.py***. This file imports the module we just created and then makes two calls to `say_hello_username():`

say.py

```
import test
usernames = ['max', 'samy', 'mahmoud']
test.say_hello_username(usernames)
```

Working With Files in Python

```
with open('data.txt', 'r') as f:
    data = f.read()
```

`open()` takes a filename and a mode as its arguments. `r` opens the file in read only mode. To write data to a file, pass in `w` as an argument instead:

```
with open('data.txt', 'w') as f:
    data = 'some data to be written to the file'
    f.write(data)
```

Getting a Directory Listing

```
>>> entries = os.listdir('my_directory/')
>>> for entry in entries:
...     print(entry)
...
...
sub_dir_c
file1.py
sub_dir_b
file3.txt
file2.csv
Sub_dir
```

References and Quizzes

a) Object Oriented Programming

1. **Python Refrance**
2. **OOP Article From DataCamp**
3. **OOP Crash corse From Real Python**
4. **Youtube Playlist**

b) Advanced Python

1. **Youtube Playlist**
2. **Methods in Python**

c) Quizzes

1. **Easy quiz**
2. **Hacker Rank Competition**

notes