Table of Contents

01-Comparison Operators

Comparison Operators

Hello Everyone, Happy New year.

Welcome to 2023 & week 4 of Learn Python with Milesh.

In this session, we will be learning about Comparison Operators in Python. These operators will allow us to compare variables and output a Boolean value (True or False).

First we'll present a table of the comparison operators and then work through some examples:

Table of Comparison Operators

In the table below, a=3 and b=4.

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Let's now work through quick examples of each of these.

Equal

In [ ]:
```
2 == 2
```

In [ ]:
```
1 == 0
```

Note that `==` is a *comparison* operator, while `=` is an *assignment* operator.

Not Equal

In [ ]:
```
2 != 1
```

In [ ]:
```
2 != 2
```

Greater Than

In [ ]:
```
2 > 1
```

In [ ]:
```
2 > 4
```

Less Than

In [ ]:
```
2 < 4
```

In [ ]:
```
2 < 1
```

Greater Than or Equal to

In [ ]:
```
2 >= 2
```

In [ ]:
```
2 >= 1
```

Less than or Equal to

In [ ]:
```
2 <= 2
```

In [ ]:
```
2 <= 4
```

Great! Go over each comparison operator to make sure you understand what each one is saying. But hopefully this was straightforward for you.

Next we will cover python statements. See you there..!!

In [ ]:
```
```

01.1-Introduction to Python Statements

Introduction to Python Statements

In this session of Learn python with Milesh, we will be doing a quick overview of Python Statements. This session will emphasize differences between Python and other languages such as C++,Java. It might feel bit different from abap.

Python vs Other Languages

Let's create a simple statement that says: "If a is greater than b, assign 2 to a and 4 to b"

Take a look at these two if statements (we will learn about building out if statements soon).

Version 1 (Other Languages)

```
if (a>b){
    a = 2;
    b = 4;
}
```

Version 2 (Python)

```
if a>b:
    a = 2
    b = 4
```

You'll notice that Python is less cluttered and much more readable than the first version. How does Python manage this?

Let's walk through the main differences:

Python gets rid of () and {} by incorporating two main factors: a *colon* and *whitespace*. The statement is ended with a colon, and whitespace is used (indentation) to describe what takes place in case of the statement.

Another major difference is the lack of semicolons or full-stop in Python. Semicolons are used to denote statement endings in many other languages, but in Python, the end of a line is the same as the end of a statement.

Lastly, to end this brief overview of differences, let's take a closer look at indentation syntax in Python vs other languages:

Indentation

Here is some pseudo-code to indicate the use of whitespace and indentation in Python:

Other Languages

```
if (x)
    if(y)
        code-statement;
else
    another-code-statement;
```

Python

```
if x:
    if y:
        code-statement
else:
    another-code-statement
```

Note how Python is so heavily driven by code indentation and whitespace. This means that code readability is a core part of the design of the Python language.

Now let's start diving deeper by coding these sort of statements in Python!

Time to code!

In [ ]:

02-if, elif, and else Statements

if, elif, else Statements

`if` Statements in Python allows us to tell the computer to perform alternative actions based on a certain set of results.

Verbally, we can imagine we are telling the computer:

"Hey if this case happens, perform some action"

We can then expand the idea further with `elif` and `else` statements, which allow us to tell the computer:

"Hey if this case happens, perform some action. Else, if another case happens, perform some other action. Else, if *none* of the above cases happened, perform this action."

Let's go ahead and look at the syntax format for `if` statements to get a better idea of this:

```
if case1:
    perform action1
elif case2:
    perform action2
else:
    perform action3
```

First Example

Let's see a quick example of this:

In [ ]:
```
if True:
    print('It was true!')
```

Let's add in some else logic:

In [ ]:
```
x = False

if x:
    print('x was True!')
else:
    print('I will be printed in any case where x is not true')
```

Multiple Branches

Let's get a fuller picture of how far `if`, `elif`, and `else` can take us!

We write this out in a nested structure. Take note of how the `if`, `elif`, and `else` line up in the code. This can help you see what `if` is related to what `elif` or `else` statements.

We'll reintroduce a comparison syntax for Python.

In [ ]:
```
loc = 'Bank'

if loc == 'Auto Shop':
    print('Welcome to the Auto Shop!')
elif loc == 'Bank':
    print('Welcome to the bank!')
else:
    print('Where are you?')
```

Note how the nested `if` statements are each checked until a True boolean causes the nested code below it to run. You should also note that you can put in as many `elif` statements as you want before you close off with an `else`.

Let's create two more simple examples for the `if`, `elif`, and `else` statements:

In [ ]:
```
user_org = 'DTAPPS'

if user_org == 'DTAPPS':
    print('Welcome to DTAPPS!')
else:
    print("Welcome, what's your Org?")
```

In [ ]:
```
person = 'George'

if person == 'Sammy':
    print('Welcome Sammy!')
elif person =='George':
    print('Welcome George!')
else:
    print("Welcome, what's your name?")
```

Indentation

It is important to keep a good understanding of how indentation works in Python to maintain the structure and order of your code. We will touch on this topic again when we start building out functions!

03-For Loops

for Loops

A `for` loop acts as an iterator in Python; it goes through items that are in a *sequence* or any other iterable item. Objects that we've learned about that we can iterate over include strings, lists, tuples, and even built-in iterables for dictionaries, such as keys or values.

We've already seen the `for` statement a little bit in past weeks but now let's formalize our understanding.

Here's the general format for a `for` loop in Python:

```
for item in object:
    statements to do stuff
```

The variable name used for the item is completely up to the coder, so use your best judgment for choosing a name that makes sense and you will be able to understand when revisiting your code. This item name can then be referenced inside your loop, for example if you wanted to use `if` statements to perform checks.

Let's go ahead and work through several example of `for` loops using a variety of data object types. We'll start simple and build more complexity later on.

Example 1

Iterating through a list

In [ ]:
```
list1 = [1,2,3,4,5,6,7,8,9,10]
```

In [ ]:
```
for num in list1:
    print(num)
```

Great! Hopefully this makes sense. Now let's add an `if` statement to check for even numbers. We'll first introduce a new concept here--the modulo.

Modulo

The modulo allows us to get the remainder in a division and uses the % symbol. For example:

In [ ]:
```
17 % 5
```

This makes sense since 17 divided by 5 is 3 remainder 2. Let's see a few more quick examples:

In [ ]:
```
# 3 Remainder 1
10 % 3
```

In [ ]:
```
# 2 Remainder 4
18 % 7
```

In [ ]:
```
# 2 no remainder
4 % 2
```

Notice that if a number is fully divisible with no remainder, the result of the modulo call is 0. We can use this to test for even numbers, since if a number modulo 2 is equal to 0, that means it is an even number!

Back to the `for` loops!

Example 2

Let's print only the even numbers from that list!

In [ ]:
```
for num in list1:
    if num % 2 == 0:
        print(num)
```

We could have also put an `else` statement in there:

In [ ]:
```
for num in list1:
    if num % 2 == 0:
        print(num)
    else:
        print('Odd number')
```

Example 3

Another common idea during a `for` loop is keeping some sort of running tally during multiple loops. For example, let's create a `for` loop that sums up the list:

In [ ]:
```
# Start sum at zero
list_sum = 0

for num in list1:
    list_sum = list_sum + num

print(list_sum)
```

Great! Read over the above cell and make sure you understand fully what is going on. Also we could have implemented a `+=` to perform the addition towards the sum. For example:

In [ ]:
```
# Start sum at zero
list_sum = 0

for num in list1:
    list_sum += num

print(list_sum)
```

Example 4

We've used `for` loops with lists, how about with strings? Remember strings are a sequence so when we iterate through them we will be accessing each item in that string.

In [ ]:
```
for letter in 'This is a string.':
    print(letter)
```

Example 5

Let's now look at how a `for` loop can be used with a tuple:

In [ ]:
```
tup = (1,2,3,4,5)

for t in tup:
    print(t)
```

Example 6

Tuples have a special quality when it comes to `for` loops. If you are iterating through a sequence that contains tuples, the item can actually be the tuple itself, this is an example of *tuple unpacking*. During the `for` loop we will be unpacking the tuple inside of a sequence and we can access the individual items inside that tuple!

In [ ]:
```
list2 = [(2,4),(6,8),(10,12)]
```

In [ ]:
```
for tup in list2:
    print(tup)
```

In [ ]:
```
# Now with unpacking!
for (t1,t2) in list2:
    print(t1)
```

Cool! With tuples in a sequence we can access the items inside of them through unpacking! The reason this is important is because many objects will deliver their iterables through tuples. Let's start exploring iterating through Dictionaries to explore this further!

Example 7

In [ ]:
```
d = {'k1':1,'k2':2,'k3':3}
```

In [ ]:
```
for item in d:
    print(item)
```

Notice how this produces only the keys. So how can we get the values? Or both the keys and the values?

We're going to introduce three new Dictionary methods: **.keys()**, **.values()** and **.items()**

In Python each of these methods return a *dictionary view object*. It supports operations like membership test and iteration, but its contents are not independent of the original dictionary – it is only a view. Let's see it in action:

In [ ]:
```
# Create a dictionary view object
d.items()
```

Since the .items() method supports iteration, we can perform *dictionary unpacking* to separate keys and values just as we did in the previous examples.

In [ ]:
```
# Dictionary unpacking
for k,v in d.items():
    print(k)
    print(v)
```

If you want to obtain a true list of keys, values, or key/value tuples, you can *cast* the view as a list:

In [ ]:
```
list(d.keys())
```

Remember that dictionaries are unordered, and that keys and values come back in arbitrary order. You can obtain a sorted list using sorted():

In [ ]:
```
sorted(d.values())
```

Conclusion

We've learned how to use for loops to iterate through tuples, lists, strings, and dictionaries. It will be an important tool for us, so make sure you know it well and understood the above examples.

More Resources

04-While Loops

while Loops

The `while` statement in Python is one of most general ways to perform iteration. A `while` statement will repeatedly execute a single statement or group of statements as long as the condition is true. The reason it is called a 'loop' is because the code statements are looped through over and over again until the condition is no longer met.

The general format of a while loop is:

```
while test:
    code statements
else:
    final code statements
```

Let's look at a few simple `while` loops in action.

In [ ]:
```
x = 0

while x < 10:
    print('x is currently: ',x)
    print(' x is still less than 10, adding 1 to x')
    x+=1
```

Notice how many times the print statements occurred and how the `while` loop kept going until the True condition was met, which occurred once x==10. It's important to note that once this occurred the code stopped. Let's see how we could add an `else` statement:

In [ ]:
```
x = 0

while x < 10:
    print('x is currently: ',x)
    print(' x is still less than 10, adding 1 to x')
    x+=1

else:
    print('All Done!')
```

break, continue, pass

We can use `break`, `continue`, and `pass` statements in our loops to add additional functionality for various cases. The three statements are defined by:

```
break: Breaks out of the current closest enclosing loop.
continue: Goes to the top of the closest enclosing loop.
pass: Does nothing at all.
```

Thinking about `break` and `continue` statements, the general format of the `while` loop looks like this:

```
while test:
    code statement
    if test:
        break
    if test:
        continue
else:
```

`break` and `continue` statements can appear anywhere inside the loop's body, but we will usually put them further nested in conjunction with an `if` statement to perform an action based on some condition.

Let's go ahead and look at some examples!

In [ ]:
```
x = 0

while x < 10:
    print('x is currently: ',x)
    print(' x is still less than 10, adding 1 to x')
    x+=1
    if x==3:
        print('x==3')
    else:
        print('continuing...')
        continue
```

Note how we have a printed statement when x==3, and a continue being printed out as we continue through the outer while loop. Let's put in a break once x ==3 and see if the result makes sense:

In [ ]:
```
x = 0

while x < 10:
    print('x is currently: ',x)
    print(' x is still less than 10, adding 1 to x')
    x+=1
    if x==3:
        print('Breaking because x==3')
        break
    else:
        print('continuing...')
        continue
```

Note how the other `else` statement wasn't reached and continuing was never printed!

After these brief but simple examples, you should feel comfortable using `while` statements in your code.

A word of caution however! It is possible to create an infinitely running loop with `while` statements. For example:

In [ ]:
```
# DO NOT RUN THIS CODE!!!!
while True:
    print("I'm stuck in an infinite loop!")
```

A quick note: If you *did* run the above cell, click on the Kernel menu above to restart the kernel!

05-List Comprehension

List Comprehensions

In addition to sequence operations and list methods, Python includes a more advanced operation called a list comprehension.

List comprehensions allow us to build out lists using a different notation. You can think of it as essentially a one line `for` loop built inside of brackets. For a simple example:

Example 1

In [ ]:
```
# Grab every letter in string
lst = [x for x in 'word']
```

In [ ]:
```
# Check
lst
```

This is the basic idea of a list comprehension. If you're familiar with mathematical notation this format should feel familiar for example: x^2 : x in { 0,1,2...10 }

Let's see a few more examples of list comprehensions in Python:

Example 2

In [ ]:
```
# Square numbers in range and turn into list
lst = [x**2 for x in range(0,11)]
```

In [ ]:
```
lst
```

Example 3

Let's see how to add in `if` statements:

In [ ]:
```
# Check for even numbers in a range
lst = [x for x in range(11) if x % 2 == 0]
```

In [ ]:
```
lst
```

Example 4

Can also do more complicated arithmetic:

In [ ]:
```
# Convert Celsius to Fahrenheit
celsius = [0,10,20.1,34.5]

fahrenheit = [((9/5)*temp + 32) for temp in celsius ]

fahrenheit
```

Example 5

We can also perform nested list comprehensions, for example:

In [ ]:
```
lst = [ x**2 for x in [x**2 for x in range(11)]]
lst
```

In [ ]:
```

```

06-Methods & Functions

Methods

We've already seen a few example of methods when learning about Object and Data Structure Types in Python. Methods are essentially functions built into objects. Later on in the community based learning in learn python with Milesh, we will learn about how to create our own objects and methods using Object Oriented Programming (OOP) and classes.

Methods perform specific actions on an object and can also take arguments, just like a function. This session will serve as just a brief introduction to methods and get you thinking about overall design methods that we will touch back upon when we reach OOP in the journey.

Methods are in the form:

```
object.method(arg1,arg2,etc...)
```

Let's take a quick look at what an example of the various methods a list has:

In [ ]:
```
# Create a simple list
lst = [1,2,3,4,5]
```

Fortunately, with iPython and the Jupyter Notebook we can quickly see all the possible methods using the tab key. The methods for a list are:

- append
- count
- extend
- insert
- pop
- remove
- reverse
- sort

Let's try out a few of them:

append() allows us to add elements to the end of a list:

In [ ]:
```
lst.append(6)
```

In [ ]:
```
lst
```

Great! Now how about count()? The count() method will count the number of occurrences of an element in a list.

In [ ]:
```
# Check how many times 2 shows up in the list
lst.count(2)
```

You can always use Shift+Tab in the Jupyter Notebook to get more help about the method. In general Python you can use the help() function:

In [ ]:
```
help(lst.count)
```

Feel free to play around with the rest of the methods for a list. Later on in this week your quiz will involve using help and Google searching for methods of different types of objects!

Great! By this session you should feel comfortable calling methods of objects in Python!

In [ ]:
```
```

Functions

Introduction to Functions

In this session of learn python with Milesh, we will consist of explaining what a function is in Python and how to create one. Functions will be one of our main building blocks when we construct larger and larger amounts of code to solve problems.

What is a function?

Formally, a function is a useful device that groups together a set of statements so they can be run more than once. They can also let us specify parameters that can serve as inputs to the functions.

On a more fundamental level, functions allow us to not have to repeatedly write the same code again and again. If you remember back to the last week sessions on strings and lists, remember that we used a function len() to get the length of a string. Since checking the length of a sequence is a common task you would want to write a function that can do this repeatedly at command.

Functions will be one of most basic levels of reusing code in Python, and it will also allow us to start thinking of program design (we will dive much deeper into the ideas of design when we learn about Object Oriented Programming).

Why even use functions?

Put simply, you should use functions when you plan on using a block of code multiple times. The function will allow you to call the same block of code without having to write it multiple times. This in turn will allow you to create more complex Python scripts. To really understand this though, we should actually write our own functions!

Function Topics

- def keyword
- simple example of a function
- calling a function with ()
- accepting parameters
- print versus return
- adding in logic inside a function
- multiple returns inside a function
- adding in loops inside a function
- tuple unpacking
- interactions between functions

def keyword

Let's see how to build out a function's syntax in Python. It has the following form:

In [ ]:
```
def name_of_function(arg1,arg2):
    '''
    This is where the function's Document String (docstring) goes.
    When you call help() on your function it will be printed out.
    '''
    # Do some stuff here
    # Return desired result
```

In [ ]:
```
name_of_function(arg1,arg2)
```

We begin with `def` then a space followed by the name of the function. Try to keep names relevant, for example len() is a good name for a length() function. Also be careful with names, you wouldn't want to call a function the same name as a built-in function in Python (such as len).

Next come a pair of parentheses with a number of arguments separated by a comma. These arguments are the inputs for your function. You'll be able to use these inputs in your function and reference them. After this you put a colon.

Now here is the important step, you must indent to begin the code inside your function correctly. Python makes use of *whitespace* to organize code. Lots of other programing languages do not do this, so keep that in mind.

Next you'll see the docstring, this is where you write a basic description of the function. Using Jupyter and Jupyter Notebooks, you'll be able to read these docstrings by pressing Shift+Tab after a function name. Docstrings are not necessary for simple functions, but it's good practice to put them in so you or other people can easily understand the code you write.

After all this you begin writing the code you wish to execute.

The best way to learn functions is by going through examples. So let's try to go through examples that relate back to the various objects and data structures we learned about before.

Simple example of a function

In [ ]:
```
def say_welcome():
    print('welcome to Learn python with Milesh')
```

Calling a function with ()

Call the function:

In [ ]:
```
say_welcome()
```

If you forget the parenthesis (), it will simply display the fact that say_hello is a function. Later on we will learn we can actually pass in functions into other functions! But for now, simply remember to call functions with ().

In [ ]:
```
say_welcome
```

Accepting parameters (arguments)

Let's write a function that greets people with their name.

In [ ]:
```
def greeting(name):
    print(f'Hello {name}')
```

In [ ]:
```
greeting('Raj')
```

Using return

So far we've only seen print() used, but if we actually want to save the resulting variable we need to use the **return** keyword.

Let's see some example that use a `return` statement. `return` allows a function to *return* a result that can then be stored as a variable, or used in whatever manner a user wants.

Example: Addition function

In [ ]:
```
def add_num(num1,num2):
    return num1+num2
```

In [ ]:
```
add_num(4,5)
```

In [ ]:
```
# Can also save as variable due to return
result = add_num(4,5)
```

In [ ]:
```
print(result)
```

What happens if we input two strings?

In [ ]:
```
add_num('one','two')
```

Very Common Question: "What is the difference between *return* and *print*?"

The return keyword allows you to actually save the result of the output of a function as a variable. The print() function simply displays the output to you, but doesn't save it for future use. Let's explore this in more detail

In [ ]:
```
def print_result(a,b):
    print(a+b)
```

In [ ]:
```
def return_result(a,b):
    return a+b
```

In [ ]:
```
print_result(10,5)
```

In [ ]:
```
# You won't see any output if you run this in a .py script
return_result(10,5)
```

But what happens if we actually want to save this result for later use?

In [ ]:
```
my_result = print_result(20,20)
```

In [ ]:
```
my_result
```

In [ ]:
```
type(my_result)
```

Be careful! Notice how print_result() doesn't let you actually save the result to a variable! It only prints it out, with print() returning None for the assignment!

In [ ]:
```
my_result = return_result(20,20)
```

In [ ]:
```
my_result
```

In [ ]:
```
my_result + my_result
```

Adding Logic to Internal Function Operations

So far we know quite a bit about constructing logical statements with Python from our previous topics from this week, such as if/else/elif statements, for and while loops, checking if an item is **in** a list or **not in** a list (Useful Operators Lecture). Let's now see how we can perform these operations within a function.

Check if a number is even

Recall the mod operator % which returns the remainder after division, if a number is even then mod 2 (% 2) should be == to zero.

In [ ]:
```
2 % 2
```

In [ ]:
```
20 % 2
```

In [ ]:
```
21 % 2
```

In [ ]:
```
20 % 2 == 0
```

In [ ]:
```
21 % 2 == 0
```

Let's use this to construct a function. Notice how we simply return the boolean check.

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

In [ ]:
```
even_check(20)
```

In [ ]:
```
even_check(21)
```

Check if any number in a list is even

Let's return a boolean indicating if **any** number in a list is even. Notice here how **return** breaks out of the loop and exits the function

In [ ]:
```
def check_even_list(num_list):
    # Go through each number
    for number in num_list:
        # Once we get a "hit" on an even number, we return True
        if number % 2 == 0:
            return True
        # Otherwise we don't do anything
        else:
            pass
```

Is this enough? NO! We're not returning anything if they are all odds!

In [ ]:
```
check_even_list([1,2,3])
```

In [ ]:
```
check_even_list([1,1,1])
```

VERY COMMON MISTAKE!! LET'S SEE A COMMON LOGIC ERROR, NOTE THIS IS WRONG!!!

In [ ]:
```
def check_even_list(num_list):
    # Go through each number
    for number in num_list:
        # Once we get a "hit" on an even number, we return True
        if number % 2 == 0:
            return True
        # This is WRONG! This returns False at the very first odd number!
        # It doesn't end up checking the other numbers in the list!
        else:
            return False
```

In [ ]:
```
# UH OH! It is returning False after hitting the first 1
check_even_list([1,2,3])
```

Correct Approach: We need to initiate a return False AFTER running through the entire loop

In [ ]:
```
def check_even_list(num_list):
    # Go through each number
    for number in num_list:
        # Once we get a "hit" on an even number, we return True
        if number % 2 == 0:
            return True
        # Don't do anything if its not even
        else:
            pass
    # Notice the indentation! This ensures we run through the entire for loop
    return False
```

In [ ]:
```
check_even_list([1,2,3])
```

In [ ]:
```
check_even_list([1,3,5])
```

Return all even numbers in a list

Let's add more complexity, we now will return all the even numbers in a list, otherwise return an empty list.

In [ ]:
```
def return_even_list(lst):
    ans = []
    for num in lst:
        if(num%2==0):
            ans.append(num)
        else:
            pass
    return ans
```

In [ ]:
```
def check_even_list(num_list):

    even_numbers = []

    # Go through each number
    for number in num_list:
        # Once we get a "hit" on an even number, we append the even number
        if number % 2 == 0:
            even_numbers.append(number)
        # Don't do anything if its not even
        else:
            pass
    # Notice the indentation! This ensures we run through the entire for loop
    return even_numbers
```

In [ ]:
```
check_even_list([1,2,3,4,5,6])
```

In [ ]:
```
check_even_list([1,3,5])
```

Returning Tuples for Unpacking

Recall we can loop through a list of tuples and "unpack" the values within them

In [ ]:
```
stock_prices = [('AMZN',200),('SAP',120),('MSFT',400)]
```

In [ ]:
```
for item in stock_prices:
    print(item)
```

In [ ]:
```
for stock,price in stock_prices:
    print(stock)
```

In [ ]:
```
for stock,price in stock_prices:
    print(price)
```

Similarly, functions often return tuples, to easily return multiple results for later use.

Let's imagine the following list:

In [ ]:
```
work_hours = [('Abby',100),('Billy',400),('Cassie',800)]
```

The employee of the month function will return both the name and number of hours worked for the top performer (judged by number of hours worked).

In [ ]:
```
def employee_check(work_hours):

    # Set some max value to intially beat, like zero hours
    current_max = 0
    # Set some empty value before the loop
    employee_of_month = ''

    for employee,hours in work_hours:
        if hours > current_max:
            current_max = hours
            employee_of_month = employee
        else:
            pass

    # Notice the indentation here
    return (employee_of_month,current_max)
```

In [ ]:
```
employee_check(work_hours)
```

Interactions between functions

Functions often use results from other functions, let's see a simple example through a guessing game. There will be 3 positions in the list, one of which is an 'O', a function will shuffle the list, another will take a player's guess, and finally another will check to see if it is correct. This is based on the classic carnival game of guessing which cup a red ball is under.

How to shuffle a list in Python

In [ ]:
```
example = [1,2,3,4,5]
```

In [ ]:
```
from random import shuffle
```

In [ ]:
```
# Note shuffle is in-place
shuffle(example)
```

In [ ]:
```
example
```

OK, let's create our simple game

In [ ]:
```
mylist = [' ','O',' ']
```

In [ ]:
```
def shuffle_list(mylist):
    # Take in list, and returned shuffle versioned
    shuffle(mylist)

    return mylist
```

In [ ]:
```
mylist
```

In [ ]:
```
shuffle_list(mylist)
```

In [ ]:
```
def player_guess():

    guess = ''

    while guess not in ['0','1','2']:

        # Recall input() returns a string
        guess = input("Pick a number: 0, 1, or 2:  ")

    return int(guess)
```

In [ ]:
```
player_guess()
```

Now we will check the user's guess. Notice we only print here, since we have no need to save a user's guess or the shuffled list.

In [ ]:
```
def check_guess(mylist,guess):
    if mylist[guess] == 'O':
        print('Correct Guess!')
    else:
        print('Wrong! Better luck next time')
        print(mylist)
```

In [ ]:
```
mylist = [' ','O',' ']
shuffle_list(mylist)

player_guess()
```

Now we create a little setup logic to run all the functions. Notice how they interact with each other!

In [ ]:
```
# Initial List
mylist = [' ','O',' ']

# Shuffle It
mixedup_list = shuffle_list(mylist)

# Get User's Guess
guess = player_guess()

# Check User's Guess
#-----------------------
# Notice how this function takes in the input
# based on the output of other functions!
check_guess(mixedup_list,guess)
```

Great! You should now have a basic understanding of creating your own functions to save yourself from repeatedly writing code! In the next session, we'll learn about how to write short functions in one line using lambda expression. See you there..!!

07-Lambda Expressions Map & Filter

Lambda Expressions, Map, and Filter

Now its time to quickly learn about two built in functions, filter and map. Once we learn about how these operate, we can learn about the lambda expression, which will come in handy when you begin to develop your skills further!

map function

The **map** function allows you to "map" a function to an iterable object. That is to say you can quickly call the same function to every item in an iterable, such as a list. For example:

In [ ]:
```
def square(num):
    return num**2
```

In [ ]:
```
my_nums = [1,2,3,4,5]
```

In [ ]:
```
map(square,my_nums)
```

In [ ]:
```
# To get the results, either iterate through map()
# or just cast to a list
list(map(square,my_nums))
```

The functions can also be more complex

In [ ]:
```
def splicer(mystring):
    if len(mystring) % 2 == 0:
        return 'even'
    else:
        return mystring[0]
```

In [ ]:
```
mynames = ['John','Cindy','Sarah','Kelly','Mike']
```

In [ ]:
```
list(map(splicer,mynames))
```

filter function

The filter function returns an iterator yielding those items of iterable for which function(item) is true. Meaning you need to filter by a function that returns either True or False. Then passing that into filter (along with your iterable) and you will get back only the results that would return True when passed to the function.

In [ ]:
```
def check_even(num):
    return num % 2 == 0
```

In [ ]:
```
nums = [0,1,2,3,4,5,6,7,8,9,10]
```

In [ ]:
```
filter(check_even,nums)
```

In [ ]:
```
list(filter(check_even,nums))
```

lambda expression

One of Pythons most useful (and for beginners, confusing) tools is the lambda expression. lambda expressions allow us to create "anonymous" functions. This basically means we can quickly make ad-hoc functions without needing to properly define a function using def.

Function objects returned by running lambda expressions work exactly the same as those created and assigned by defs. There is key difference that makes lambda useful in specialized roles:

lambda's body is a single expression, not a block of statements.

- The lambda's body is similar to what we would put in a def body's return statement. We simply type the result as an expression instead of explicitly returning it. Because it is limited to an expression, a lambda is less general that a def. We can only squeeze design, to limit program nesting. lambda is designed for coding simple functions, and def handles the larger tasks.

Lets slowly break down a lambda expression by deconstructing a function:

In [ ]:
```
def square(num):
    result = num**2
    return result
```

In [ ]:
```
square(2)
```

We could simplify it:

In [ ]:
```
def square(num):
    return num**2
```

In [ ]:
```
square(2)
```

We could actually even write this all on one line.

In [ ]:
```
def square(num): return num**2
```

In [ ]:
```
square(2)
```

This is the form a function that a lambda expression intends to replicate. A lambda expression can then be written as:

In [ ]:
```
lambda num: num ** 2
```

In [ ]:
```
# You wouldn't usually assign a name to a lambda expression, this is just for demonstration!
square = lambda num: num **2
```

In [ ]:
```
square(2)
```

So why would use this? Many function calls need a function passed in, such as map and filter. Often you only need to use the function you are passing in once, so instead of formally defining it, you just use the lambda expression. Let's repeat some of the examples from above with a lambda expression

In [ ]:
```
list(map(lambda num: num ** 2, my_nums))
```

In [ ]:
```
list(filter(lambda n: n % 2 == 0,nums))
```

Here are a few more examples, keep in mind the more comples a function is, the harder it is to translate into a lambda expression, meaning sometimes its just easier (and often the only way) to create the def keyword function.

Lambda expression for grabbing the first character of a string:

In [ ]:
```
lambda s: s[0]
```

Lambda expression for reversing a string:

In [ ]:
```
lambda s: s[::-1]
```

You can even pass in multiple arguments into a lambda expression. Again, keep in mind that not every function can be translated into a lambda expression.

In [ ]:
```
lambda x,y : x + y
```

You will find yourself using lambda expressions often with certain non-built-in libraries, for example the pandas library for data analysis works very well with lambda expressions.

Assignment Questions:

1. **Problem Statement:**

Given is a list of numbers, return the sum of the numbers which are in odd indices (considering that the list starts with index 0)

For example: lst = [25,245,58,45,15,85,59,45] then your output should be 420.

In this case, the values at odd indexes are 245, 45, 85, 45.

Input Specification:

input1: A list of numbers.

Output Specification:

Integer value

2. **Problem Statement:**

A company has its product IDs in a list and they want to get the list of only those products IDs whose ID starts with ‘a’ and has a total of 6 (including ‘a’) alpha-numeric characters.

Input Specification:

input1: List of the product IDs.

Sample Input: ["a25648","a5678","457895","65358"]

Output Specification:

Return a list of product IDs that start with 'a' and has a total of 6 (including 'a') alpha-numeric characters.

Sample Output: ['a25648']

3. **Problem Statement:**

Given a String input, calculate and print the number of upper-case letters and lower case letters.

Sample String: 'Hello Mr. Rogers, how are you this fine Tuesday?'

Expected Output:
No. of Upper case characters: 4
No. of Lower case Characters: 33

HINT: Two string methods that might prove useful: .isupper() and .islower()

4. **Problem Statement:**

Two friends (Reetesh, Sumit) went to a restaurant to have dinner. Since Reetesh was the manager of Sumit, Reetesh decided to pay 70% of the total bill while Sumit would pay 30% of the total bill.

Given is the dictionary of key-value pair containing item_ordered:price mapping.

For example: orderbill = {"chicken65":300,"plaindosa":120}

Print the amount that has to paid by Reetesh and Sumit

Input Specification:

input1: a dictionary of key-value pair containing {"item_ordered":price} mapping.

Output Specification:

Return a dictionary of key-value pair containing {"reetesh":amount,"sumit":amount} where amount (float) is the money they are going to pay.

5. **Problem Statement:**

Given is a string of lyrics of a song. Find out how many times the word “welcome” occurs in the lyrics.

For example: If given string of lyrics is "I welcome you all to Learn Python along with milesh, welcome you all to our team" then the output should be 2 (integer).

Input Specification:

A string of lyrics.

Output Specification:

Integer specifying No. of times word "welcome" appears in the script.

Please feel free to reach out in case of any query to:

- Rajashekaran, Sreejith
- Jain, Rajat
- Mallick, Prabir Kumar

Good Luck!