Conditions and Loops with Python Iterables

Week 4

Python Conditions and If statements

- Python supports the usual logical conditions from mathematics:

Equals: a == b

Not Equals: a != b

Less than: a < b

Less than or equal to: a <= b

Greater than: a > b

Greater than or equal to: a >= b

These conditions can be used in several ways, most commonly in "if statements" and loops.

An "if statement" is written by using the if keyword.

If Else

- In this example we use two variables, **a** and **b**, which are used as part of the **if statement** to test whether **b** is **greater than (>)** **a**.

```
a = 33
b = 200
if b > a:
        print("b is greater than a")
```

Output: **b** is greater than **a**

This is because **b** > **a** = True

As **a** is **33**, and **b** is **200**, we know that **200** is **greater than (>)** **33**, and so we print to screen that **"b is greater than a"**.

If statement, without indentation will raise an error, IndentationError: expected an indented block

Elif

- The **elif** keyword is pythons' way of saying "if the previous conditions were not true, then try this condition".

```
a = 33
b = 33
if b > a:
		print("b is greater than a")
elif a == b:
		print("a and b are equal")
```

In this example **a** is equal to **b**, so the first condition is not true, but the **elif** condition is TRUE, so we print to screen that "**a** and **b** are equal".

Else

- The **else** keyword catches anything which isn't caught by the preceding conditions.

```
a = 200
b = 33
if b > a:
        print("b is greater than a")
elif a == b:
        print("a and b are equal")
else:
        print("a is greater than b")
```

In this example a is **greater than** b, so the first condition is **NOT TRUE**, also the elif condition is **NOT TRUE**, so we go to the **else** condition and print to screen that "**a** is greater than **b**".

You can also have an else without the elif:

Short Hand If

- if you have only one statement to execute, you can put it on the same line as the if statement.

```
if a > b: print("a is greater than b")
```

Short Hand If ... Else

- If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

```
a = 2
b = 330
print("A") if a > b else print("B")
```

You can also have multiple else statements on the same line:

```
a = 330
b = 330
print("A") if a > b else print("=") if a == b else print("B")
```

And

- The and keyword is a logical operator, and is used to combine conditional statements:

- Test if a is greater than b, AND if c is greater than a:

```
a = 200
b = 33
c = 500
if a > b and c > a:
        print("Both conditions are True")
```

Output: Both conditions are True

Or

- The or keyword is a logical operator, and is used to combine conditional statements:

- Test if a is greater than b, OR if a is greater than c:

```
a = 200
b = 33
c = 500
if a > b or a > c:
        print("At least one of the conditions is True")
```

Output: At least one of the conditions is True

Nested If

- You can have if statements inside if statements, this is called nested if statements.

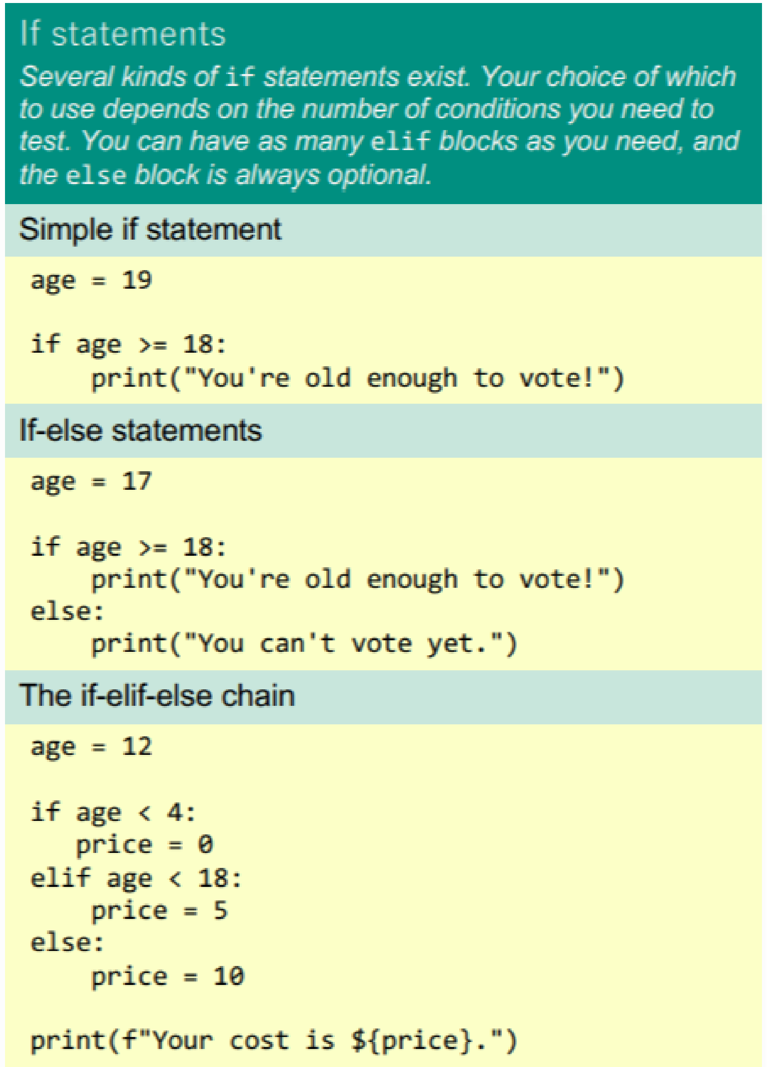
```
x = 41

if x > 10:
        print("Above ten,")
        if x > 20:
                print("and also above 20!")
        else:
                print("but not above 20.")
```

The pass Statement

- if statements cannot be empty, but if you for some reason have an if statement with no content, put in the pass statement to avoid getting an error.

```
a = 33
b = 200

if b > a:
        pass
```

Python If statement snippets



If statements

Several kinds of if statements exist. Your choice of which to use depends on the number of conditions you need to test. You can have as many elif blocks as you need, and the else block is always optional.

Simple if statement

```python
age = 19

if age >= 18:
    print("You're old enough to vote!")
```

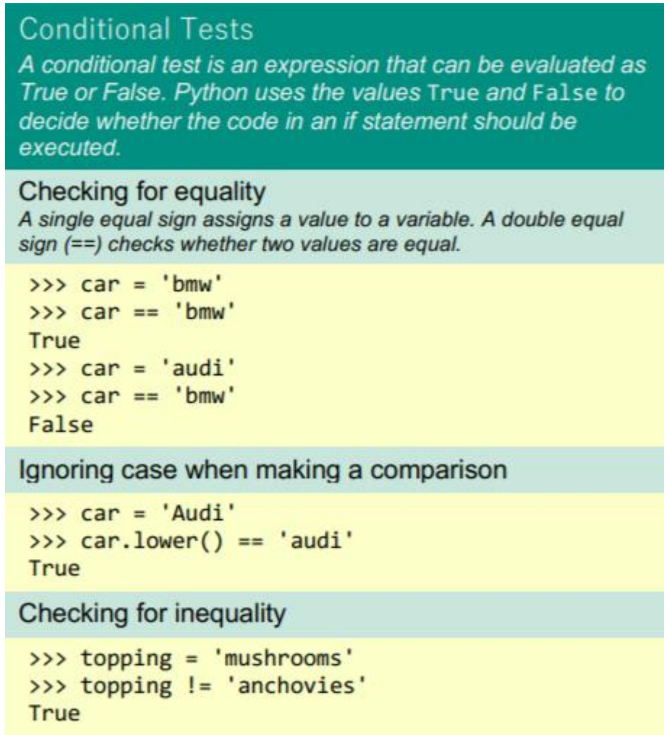
If-else statements

```python
age = 17

if age >= 18:
    print("You're old enough to vote!")
else:
    print("You can't vote yet.")
```

The if-elif-else chain

```python
age = 12

if age < 4:
   price = 0
elif age < 18:
    price = 5
else:
    price = 10

print(f"Your cost is ${price}.")
```

Python Condition Snippets



Conditional Tests

A conditional test is an expression that can be evaluated as True or False. Python uses the values `True` and `False` to decide whether the code in an if statement should be executed.

Checking for equality

A single equal sign assigns a value to a variable. A double equal sign (==) checks whether two values are equal.

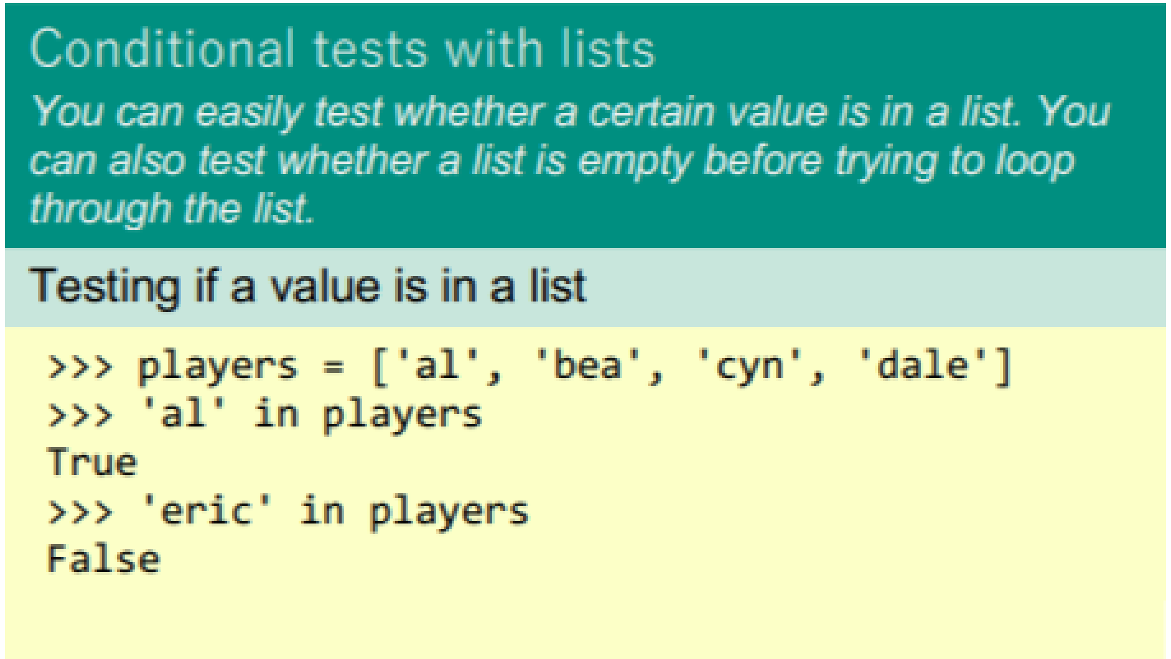
```
>>> car = 'bmw'
>>> car == 'bmw'
True
>>> car = 'audi'
>>> car == 'bmw'
False
```

Ignoring case when making a comparison

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
```

Checking for inequality

```
>>> topping = 'mushrooms'
>>> topping != 'anchovies'
True
```



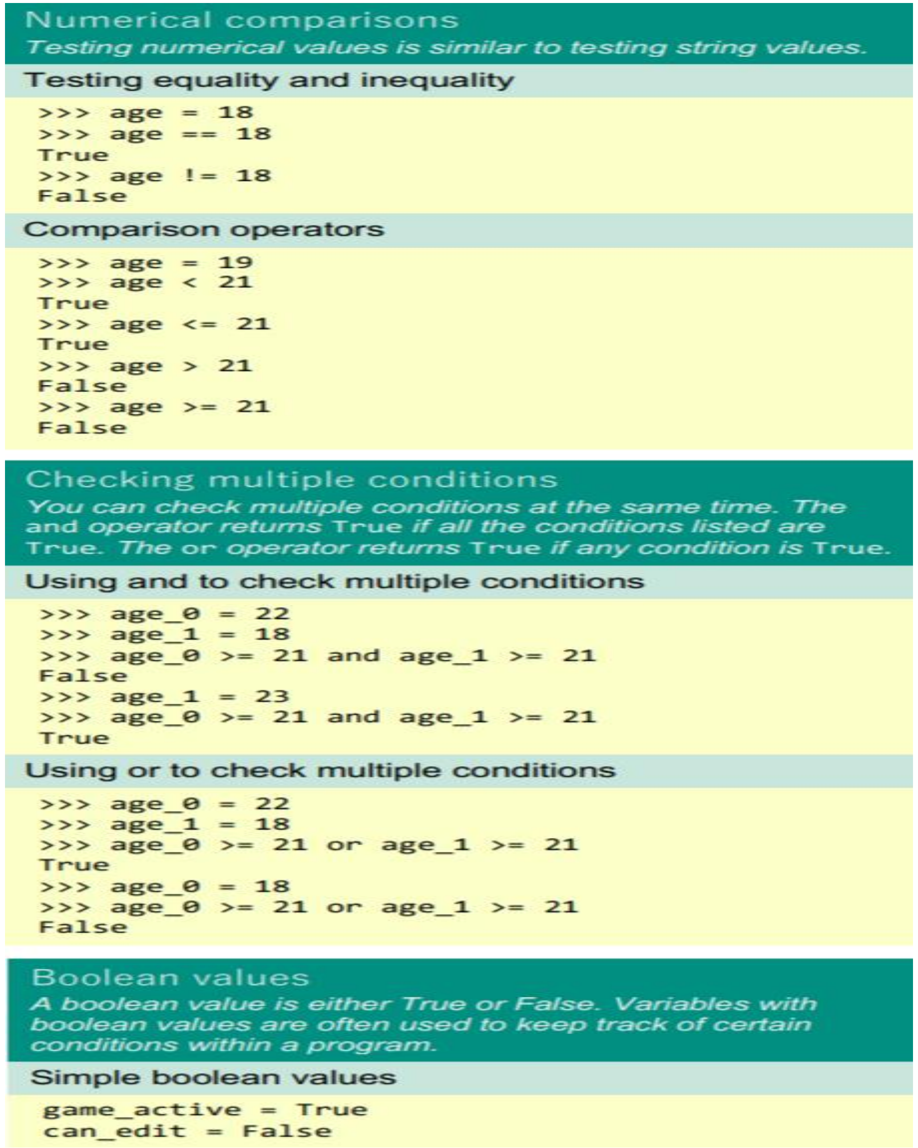
Conditional tests with lists

You can easily test whether a certain value is in a list. You can also test whether a list is empty before trying to loop through the list.

Testing if a value is in a list

```
>>> players = ['al', 'bea', 'cyn', 'dale']
>>> 'al' in players
True
>>> 'eric' in players
False
```

More snippets



Numerical comparisons

Testing numerical values is similar to testing string values.

Testing equality and inequality

```
>>> age = 18
>>> age == 18
True
>>> age != 18
False
```

Comparison operators

```
>>> age = 19
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

Checking multiple conditions

You can check multiple conditions at the same time. The and *operator returns* True *if all the conditions listed are* True*. The* or *operator returns* True *if any condition is* True*.*

Using and to check multiple conditions

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 and age_1 >= 21
False
>>> age_1 = 23
>>> age_0 >= 21 and age_1 >= 21
True
```

Using or to check multiple conditions

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 or age_1 >= 21
True
>>> age_0 = 18
>>> age_0 >= 21 or age_1 >= 21
False
```

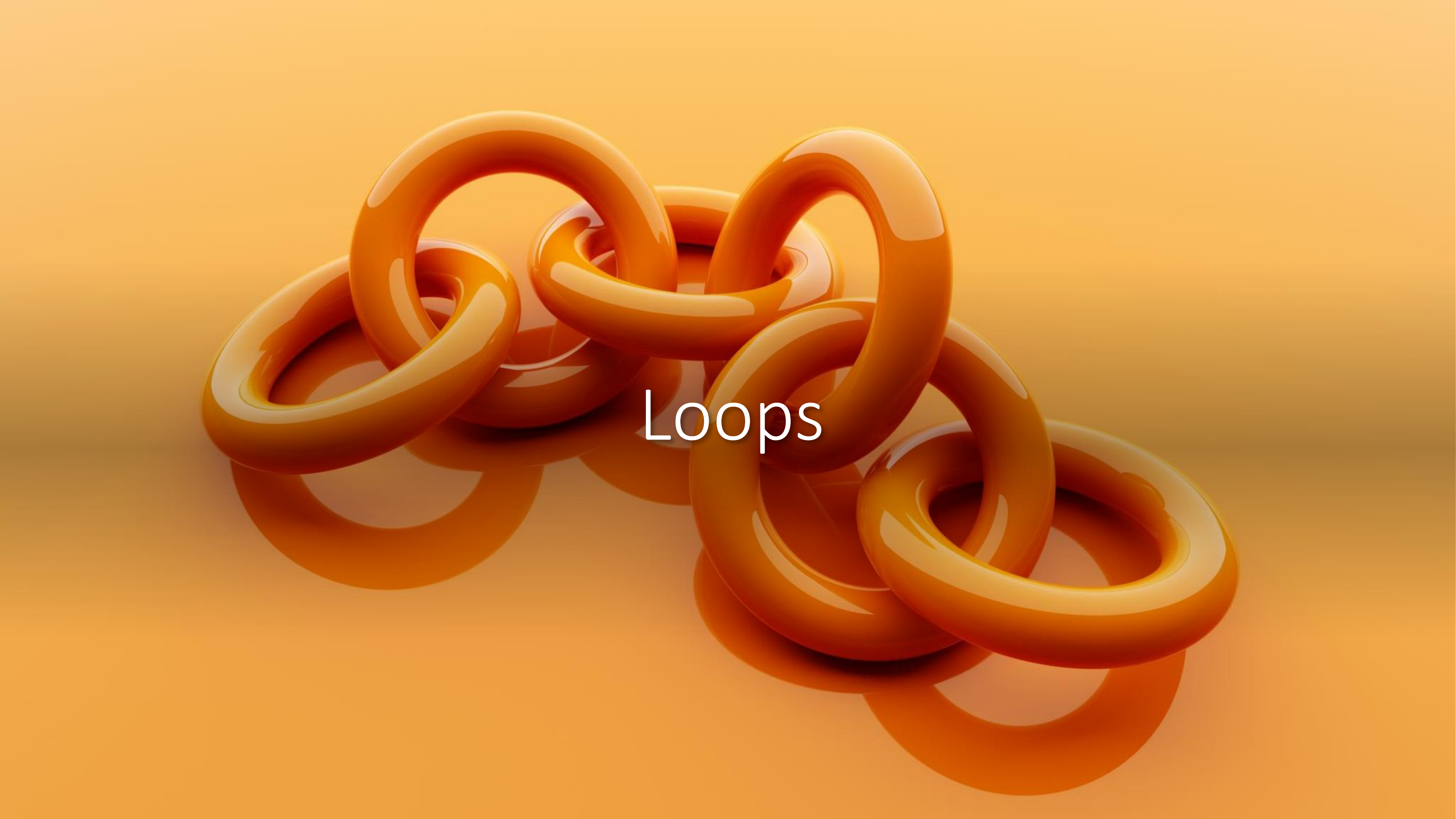
Boolean values

A boolean value is either True or False. Variables with boolean values are often used to keep track of certain conditions within a program.

Simple boolean values

```
game_active = True
can_edit = False
```

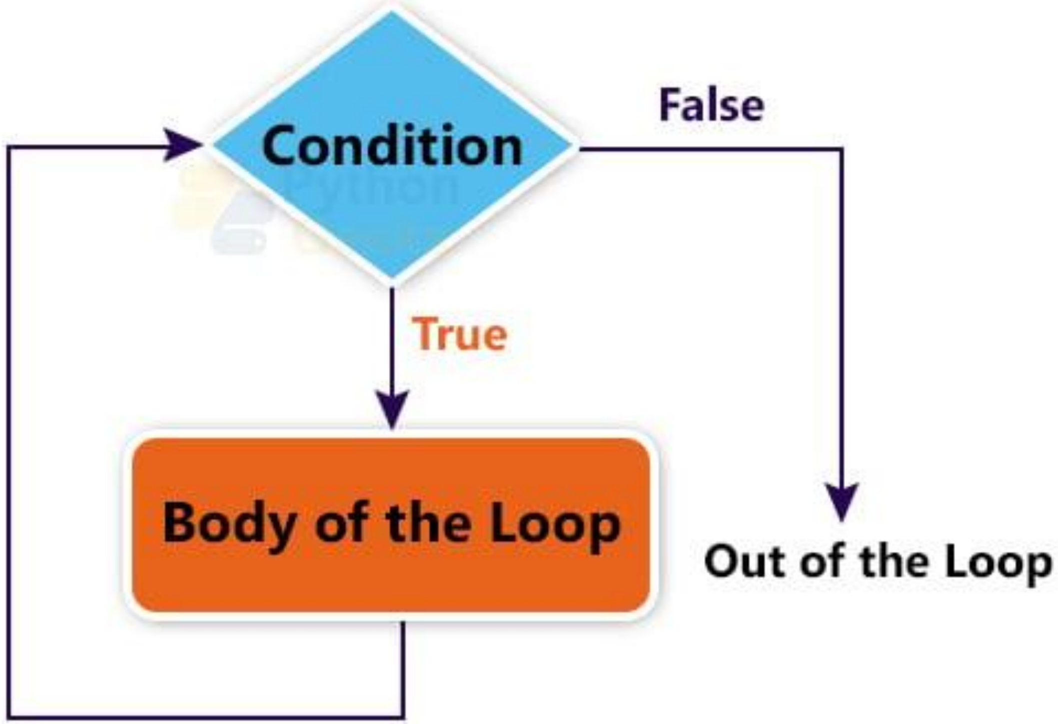
Loops



While Loops

While loop flow chart

- While loops execute a set of lines of code iteratively till a condition is satisfied. Once the condition results in False, it stops execution, and the part of the program after the loop starts executing.



While Loops

- With the while loop we can execute a set of statements as long as a condition is true.

- Print i as long as i is less than 6:

```
i = 1
while i < 6:
        print(i)
        i += 1
```

Note: remember to increment i, or else the loop will continue forever.

The while loop requires relevant variables to be ready, in this example we need to define an indexing variable, i, which we set to 1.

The break Statement

- With the break statement we can stop the loop even if the while condition is true:

Exit the loop when i is 3:

```
i = 1
while i < 6:
    print(i)
    if i == 3:
        break
    i += 1
```

The continue Statement

With the continue statement we can stop the current iteration, and continue with the next:

Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
    i += 1
    if i == 3:
        continue
    print(i)
```

The else Statement

- With the else statement we can run a block of code once when the condition no longer is true:

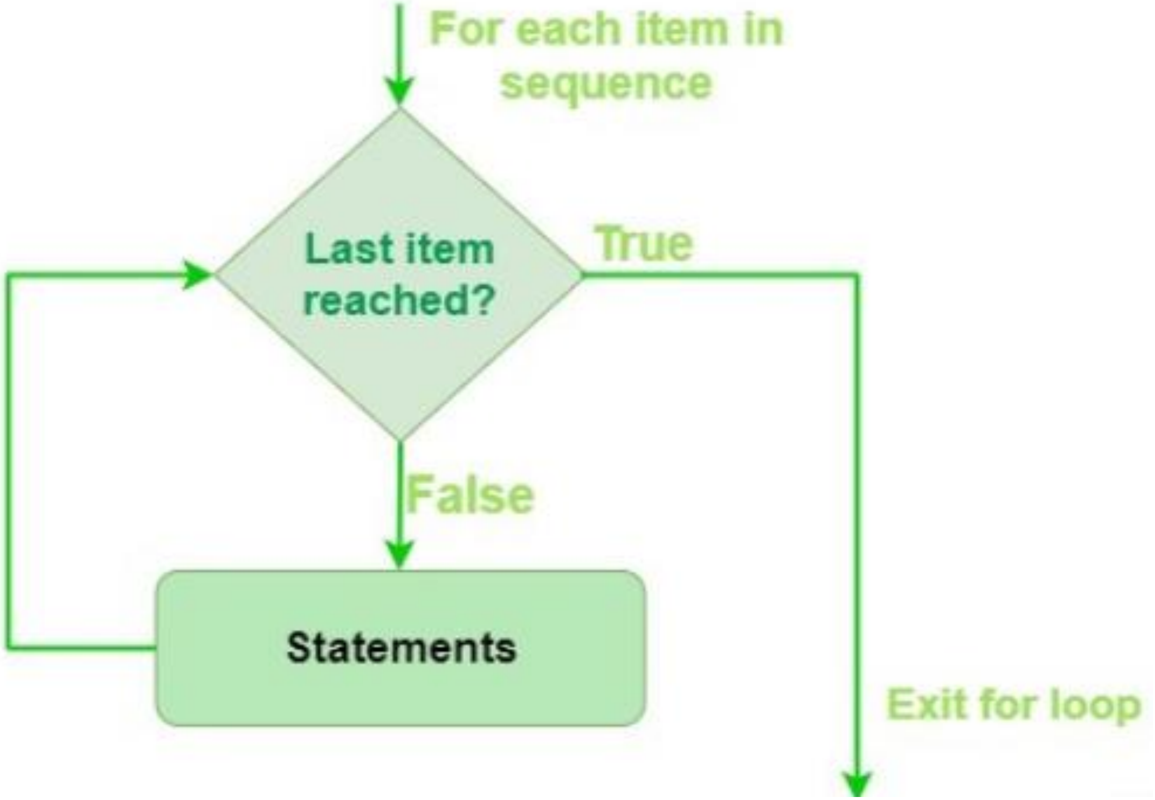
- Print a message once the condition is false:

```
i = 1
while i < 6:
        print(i)
        i += 1
else:
        print("i is no longer less than 6")
```

For Loops

The for loop flowchart

Here the iterable is a collection of objects like lists, and tuples. The indented statements inside the for loops are executed once for each item in an iterable. The variable var takes the value of the next item of the iterable each time through the loop.



Iterables

- An object which can be looped over or iterated over with the help of a for loop.

- Familiar examples of iterables include lists, tuples, and strings - any such sequence data types can be iterated over in a for-loop.

- Such include a string.

Python For Loops

- Use for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).
- Less like the for keyword in other programming languages and works more like an iterator method as found in other object-orientated programming languages.
- it can execute a set of statements, once for each item in a list, tuple, set etc.

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
        print(x)
```

Output:

```
apple
banana
cherry
```

The for loop does not require an indexing variable to set beforehand.

Looping Through a String

- Even strings are iterable objects, they contain a sequence of characters:

```
for x in "banana":
        print(x)
```

Output:

```
b
a
n
a
n
a
```

The break Statement

- With the break statement we can stop the loop before it has looped through all the items:

- Exit the loop when x is "banana":

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
        print(x)
        if x == "banana":
                break
```

Output:

The break Statement

- Exit the loop when x is "banana", but this time the break comes before the print:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
        if x == "banana":
            break
    print(x)
```

Output:

The continue Statement

- With the continue statement we can stop the current iteration of the loop, and continue with the next:

- Do not print banana:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
        if x == "banana":
                continue
        print(x)
```

Output:

The range() Function

- To loop through a set of code a specified number of times, we can use the range() function,
- The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

- Using the range() function:

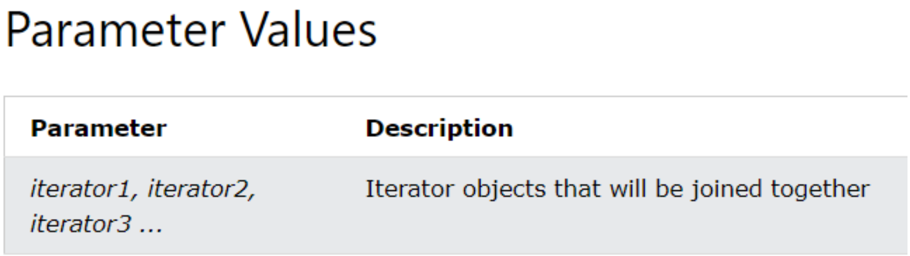
```
for x in range(6):
        print(x)
```

Python Zip

- The zip() function returns a zip object, which is an iterator of tuples where the first item in each passed iterator is paired together, and then the second item in each passed iterator are paired together etc.

- If the passed iterators have different lengths, the iterator with the least items decides the length of the new iterator.

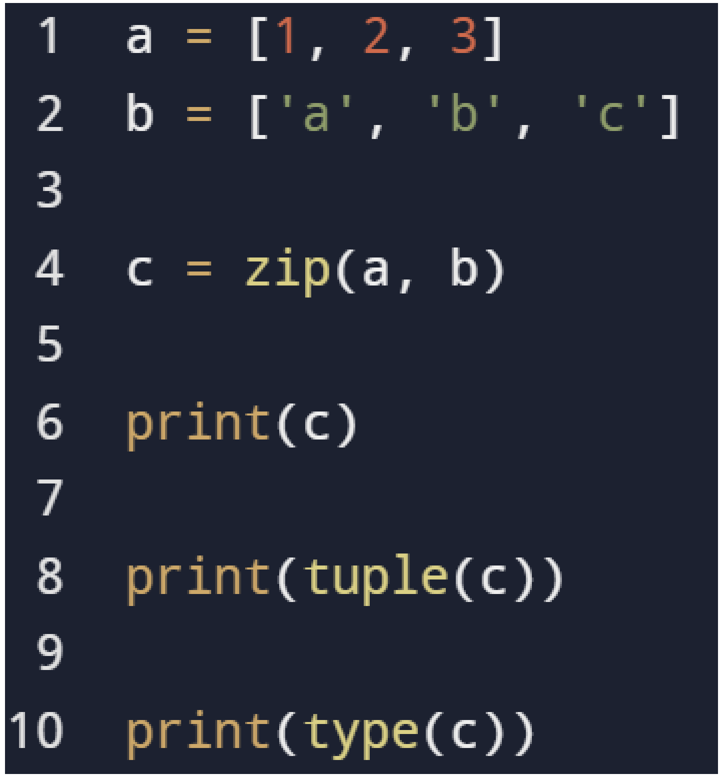
- `zip(iterator1, iterator2, iterator3 ...)`



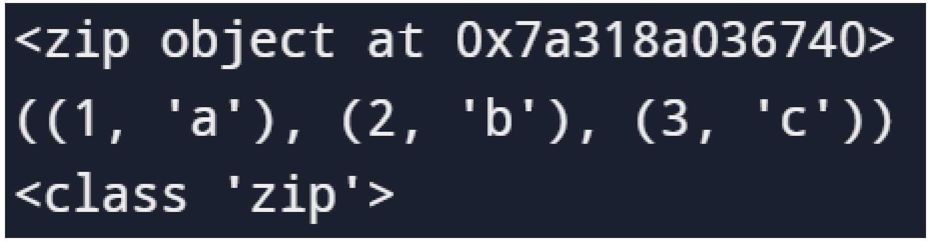
Parameter Values

Parameter	Description
iterator1, iterator2, iterator3 ...	Iterator objects that will be joined together

Printing Zip

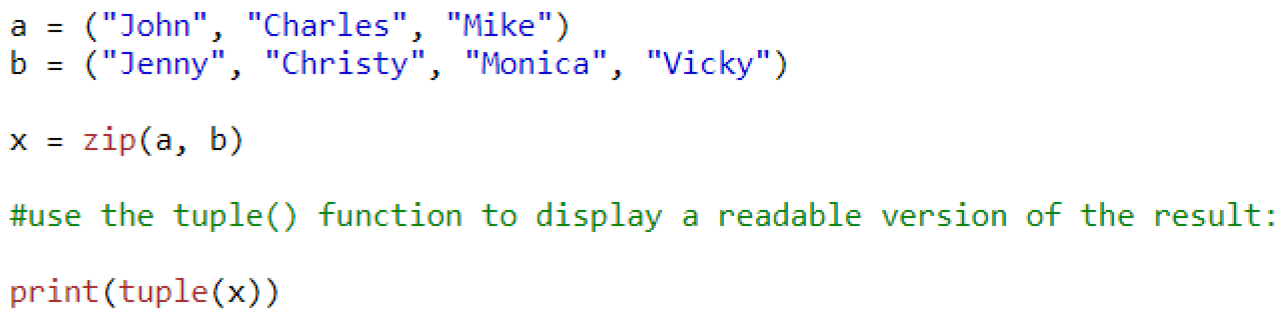


```python
a = [1, 2, 3]
b = ['a', 'b', 'c']

c = zip(a, b)

print(c)

print(tuple(c))

print(type(c))
```

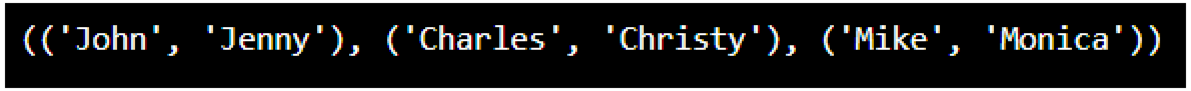


```
<zip object at 0x7a318a036740>
((1, 'a'), (2, 'b'), (3, 'c'))
<class 'zip'>
```

Zip Example



```
a = ("John", "Charles", "Mike")
b = ("Jenny", "Christy", "Monica", "Vicky")

x = zip(a, b)

#use the tuple() function to display a readable version of the result:

print(tuple(x))
```

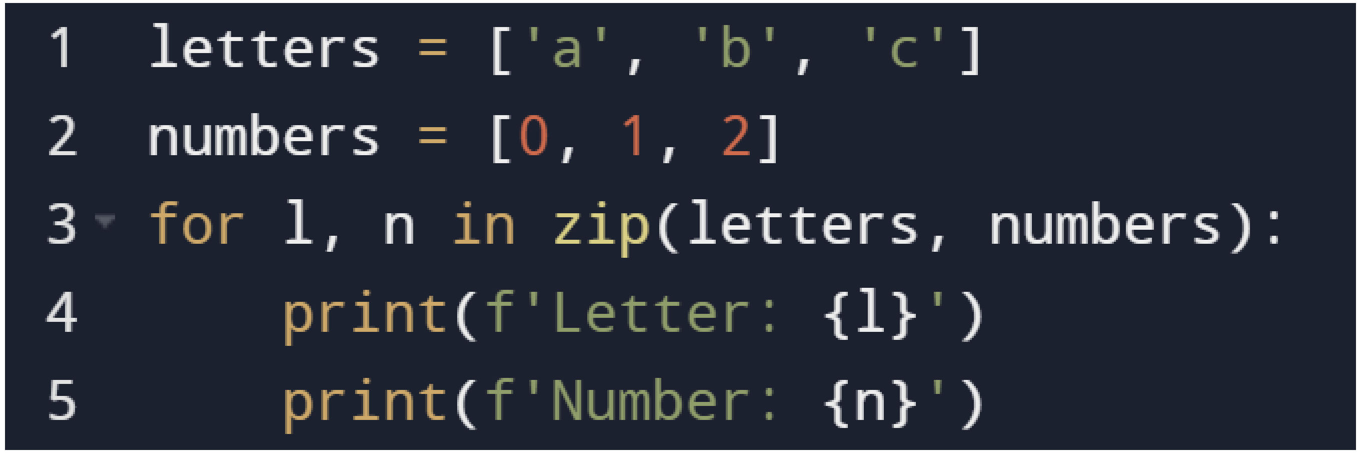


```
(('John', 'Jenny'), ('Charles', 'Christy'), ('Mike', 'Monica'))
```

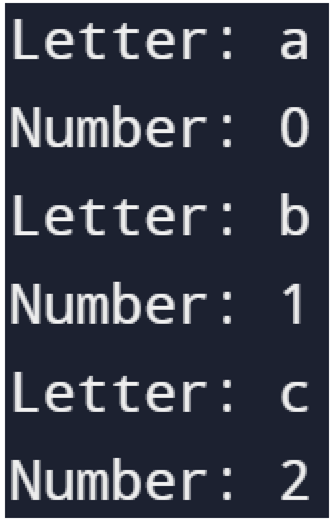
Returns a **zip object**, which is an **iterator of tuples** where the **first item in each passed iterator is paired together**, and then **the second item in each passed iterator are paired together** etc.

List Traversal

- Python’s zip() function allows you to iterate in parallel over two or more iterables. Since zip() generates tuples, you can unpack these in the header of a for loop:



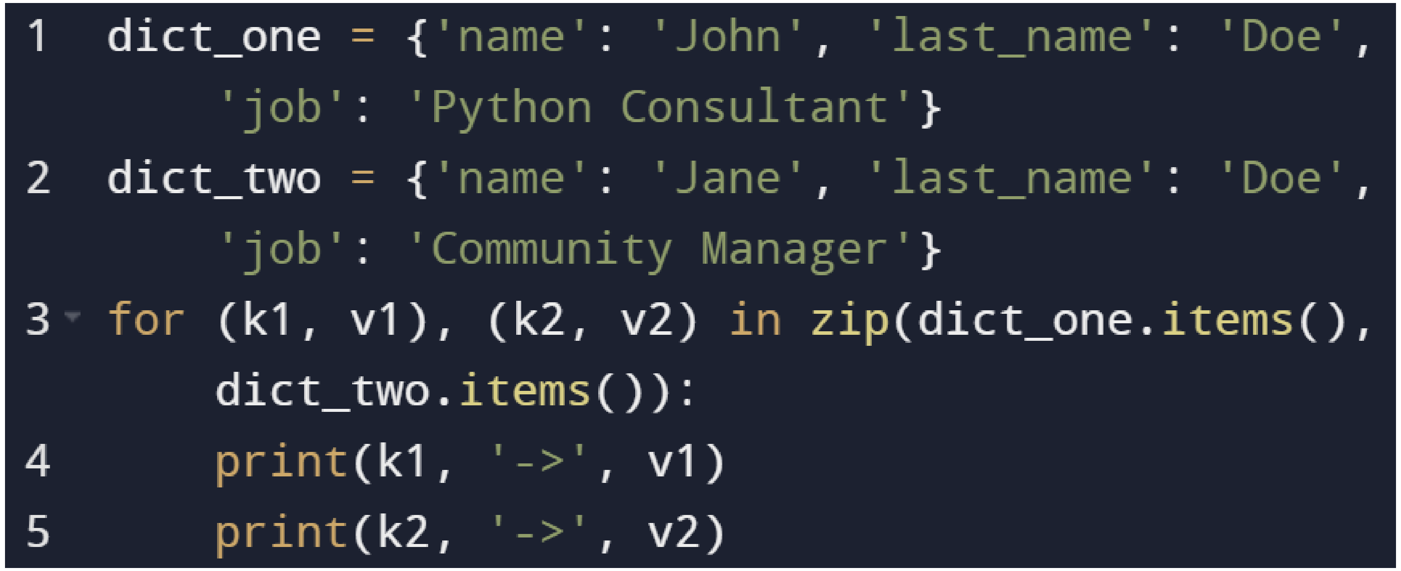
```python
letters = ['a', 'b', 'c']
numbers = [0, 1, 2]
for l, n in zip(letters, numbers):
    print(f'Letter: {l}')
    print(f'Number: {n}')
```



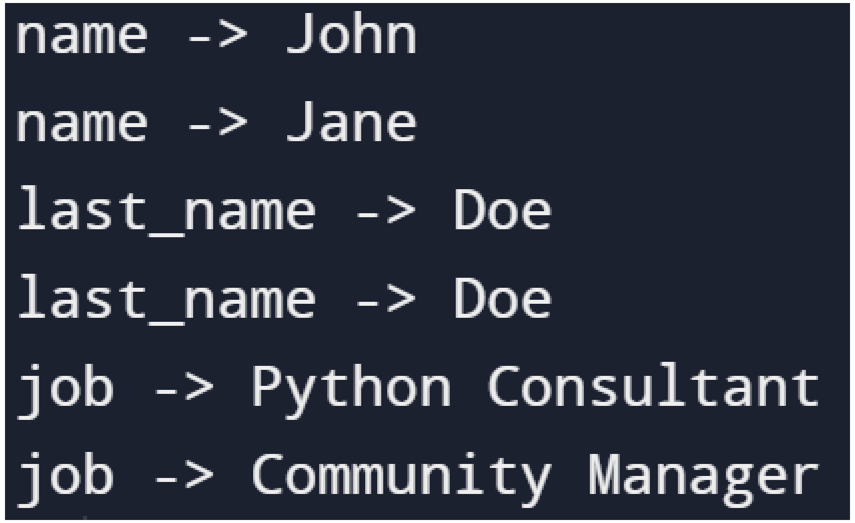
```
Letter: a
Number: 0
Letter: b
Number: 1
Letter: c
Number: 2
```

Traversing Dictionaries in Parallel

In Python 3.6 and beyond, dictionaries are ordered collections, meaning they keep their elements in the same order in which they were introduced. If you take advantage of this feature, then you can use the Python zip() function to iterate through multiple dictionaries in a safe and coherent way:



```python
dict_one = {'name': 'John', 'last_name': 'Doe',
    'job': 'Python Consultant'}
dict_two = {'name': 'Jane', 'last_name': 'Doe',
    'job': 'Community Manager'}
for (k1, v1), (k2, v2) in zip(dict_one.items(),
        dict_two.items()):
    print(k1, '->', v1)
    print(k2, '->', v2)
```



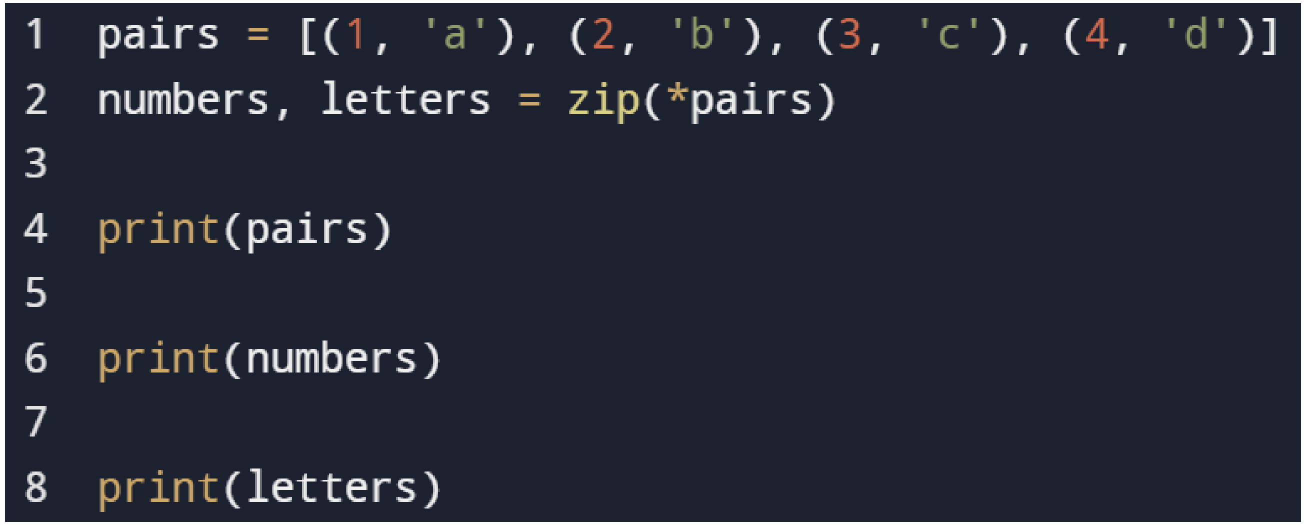
```
name -> John
name -> Jane
last_name -> Doe
last_name -> Doe
job -> Python Consultant
job -> Community Manager
```

Unzipping a Sequence

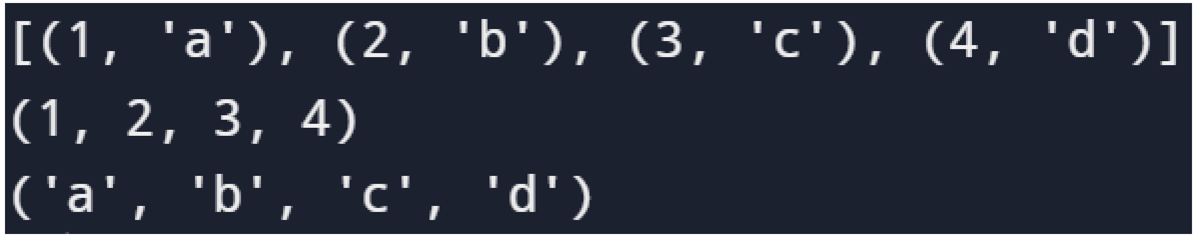
“If there’s a zip() function, then why is there no unzip() function that does the opposite?”

The reason why there’s no unzip() function in Python is because the opposite of zip() is… well, zip(). Do you recall that the Python zip() function works just like a real zipper? The examples so far have shown you how Python zips things closed. So, how do you unzip Python objects?

Say you have a list of tuples and want to separate the elements of each tuple into independent sequences. To do this, you can use zip() along with the unpacking operator *, like so:



```python
pairs = [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
numbers, letters = zip(*pairs)

print(pairs)

print(numbers)

print(letters)
```



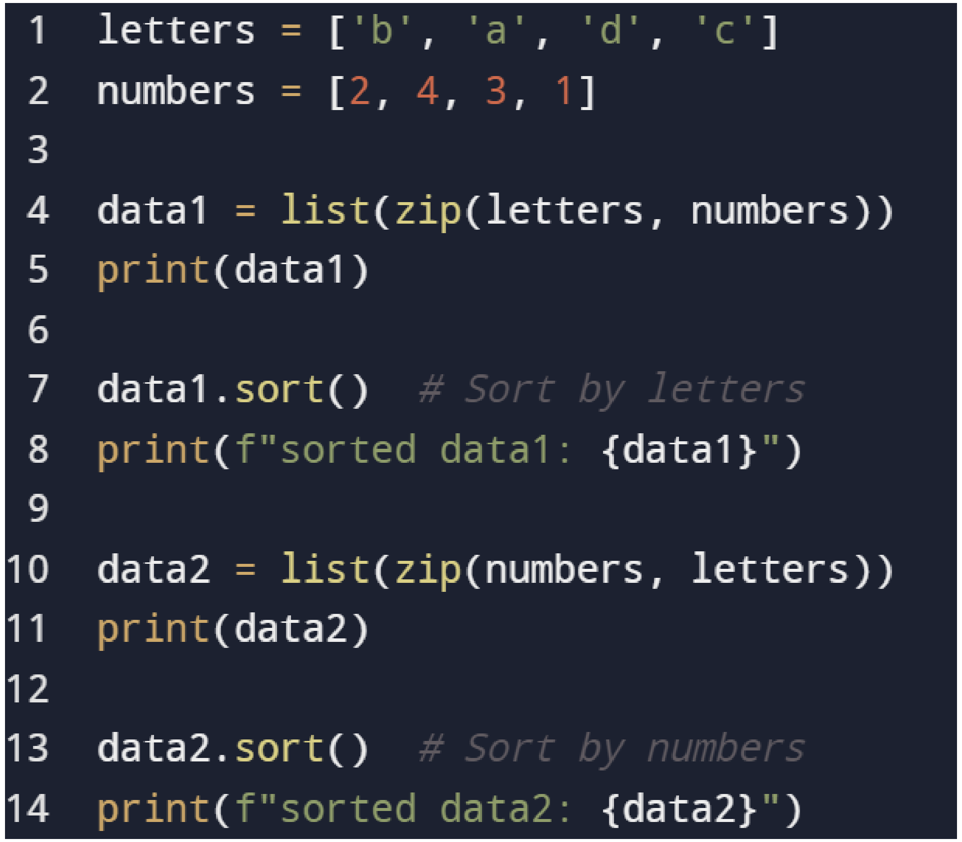
```
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
(1, 2, 3, 4)
('a', 'b', 'c', 'd')
```

Parallel Sorting

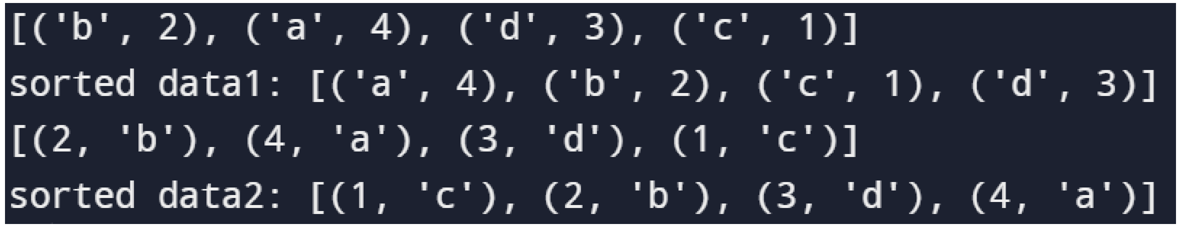
Sorting is a common operation in programming. Suppose you want to combine two lists and sort them at the same time. To do this, you can use zip() along with .sort() as follows

sorted() runs through the iterator generated by zip() and sorts the items by letters, all in one go. This approach can be a little bit faster since you'll need only two function calls: zip() and sorted().

With sorted(), you're also writing a more general piece of code. This will allow you to sort any kind of sequence, not just lists.



```python
letters = ['b', 'a', 'd', 'c']
numbers = [2, 4, 3, 1]

data1 = list(zip(letters, numbers))
print(data1)

data1.sort()  # Sort by letters
print(f"sorted data1: {data1}")

data2 = list(zip(numbers, letters))
print(data2)

data2.sort()  # Sort by numbers
print(f"sorted data2: {data2}")
```



```
[('b', 2), ('a', 4), ('d', 3), ('c', 1)]
sorted data1: [('a', 4), ('b', 2), ('c', 1), ('d', 3)]
[(2, 'b'), (4, 'a'), (3, 'd'), (1, 'c')]
sorted data2: [(1, 'c'), (2, 'b'), (3, 'd'), (4, 'a')]
```

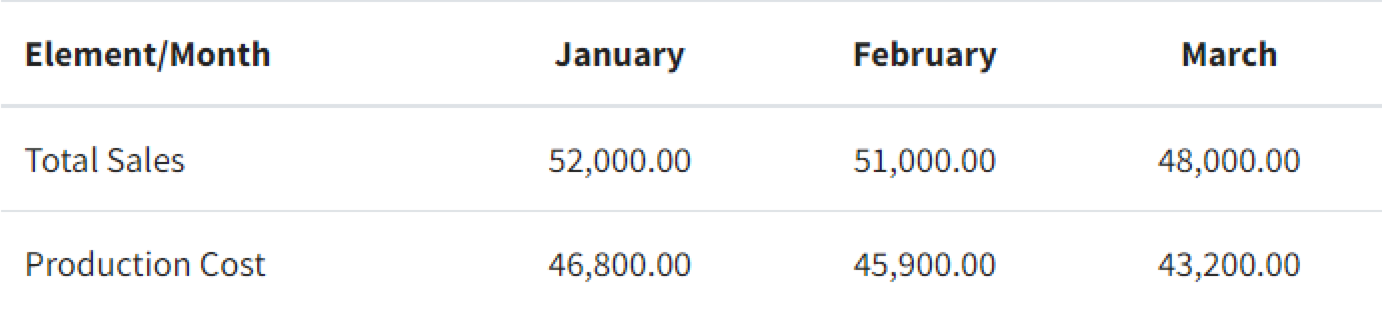
Pair Calculation

You can use the Python zip() function to make some quick calculations. Suppose you have the following data in a spreadsheet.

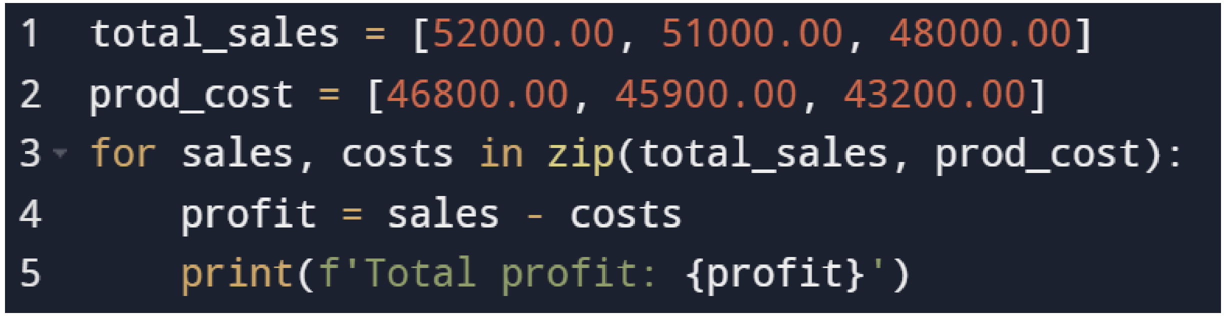
You're going to use this data to calculate your monthly profit. zip() can provide you with a fast way to make the calculations.

The example calculates the profit for each month by subtracting costs from sales.

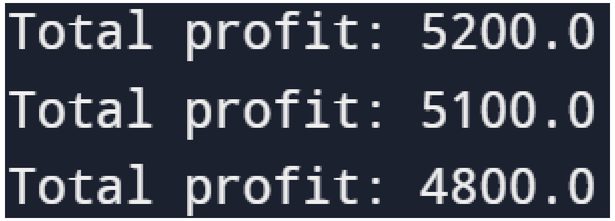
Python's zip() function combines the right pairs of data to make the calculations. You can generalize this logic to make any kind of complex calculation with the pairs returned by zip()



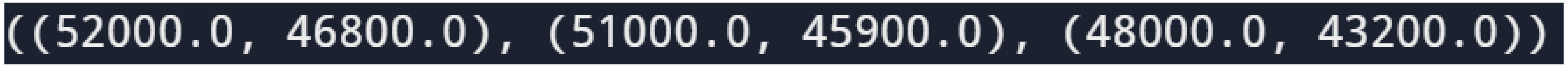
Element/Month	January	February	March
Total Sales	52,000.00	51,000.00	48,000.00
Production Cost	46,800.00	45,900.00	43,200.00



```python
total_sales = [52000.00, 51000.00, 48000.00]
prod_cost = [46800.00, 45900.00, 43200.00]
for sales, costs in zip(total_sales, prod_cost):
    profit = sales - costs
    print(f'Total profit: {profit}')
```



```
Total profit: 5200.0
Total profit: 5100.0
Total profit: 4800.0
```

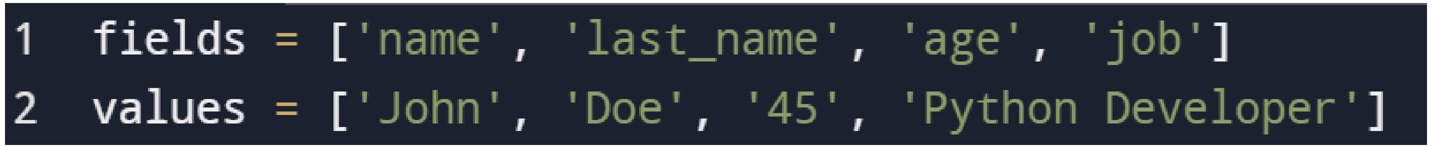


```
((52000.0, 46800.0), (51000.0, 45900.0), (48000.0, 43200.0))
```

Dictionary Building with Zip

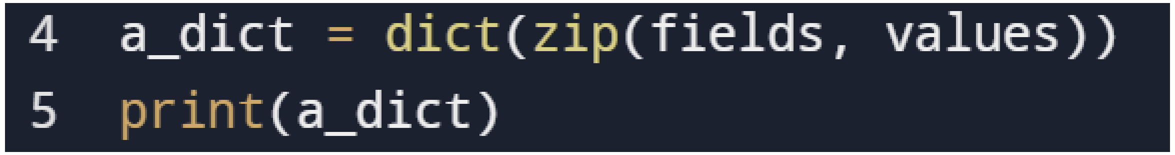
Python’s dictionaries are a very useful data structure.

Sometimes, you might need to build a dictionary from two different but closely related sequences. A convenient way to achieve this is to use dict() and zip() together. For example, suppose you retrieved a person’s data from a form or a database. Now you have the following lists of data.

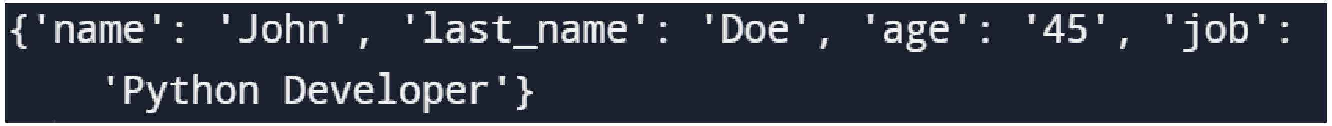


```
fields = ['name', 'last_name', 'age', 'job']
values = ['John', 'Doe', '45', 'Python Developer']
```

You need to create a dictionary for further processing. In this case, you can use dict() along with zip() as follows:



```
a_dict = dict(zip(fields, values))
print(a_dict)
```



```
{'name': 'John', 'last_name': 'Doe', 'age': '45', 'job':
    'Python Developer'}
```

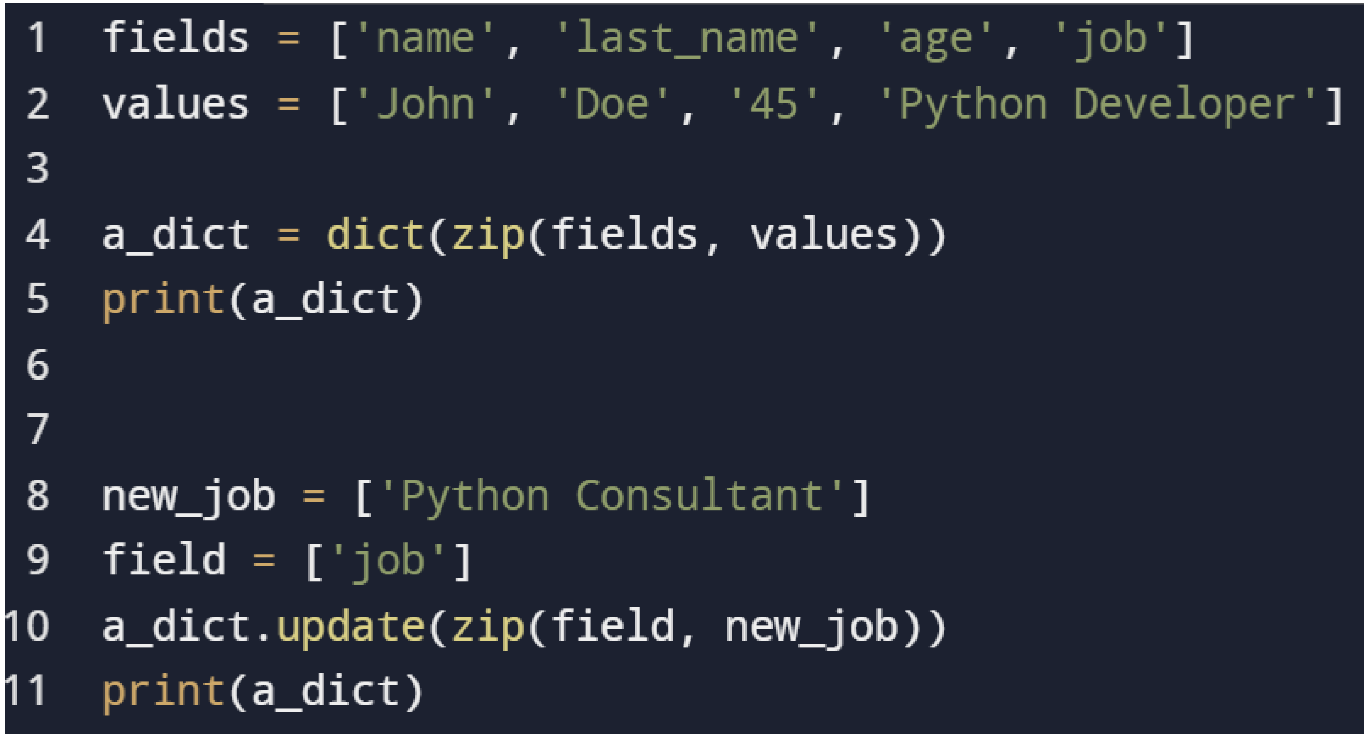
Dynamic Data with Dict and Zip

Here, you create a dictionary that combines the two lists. zip(fields, values) returns an iterator that generates 2-items tuples. If you call dict() on that iterator, then you'll be building the dictionary you need. The elements of fields become the dictionary's keys, and the elements of values represent the values in the dictionary.

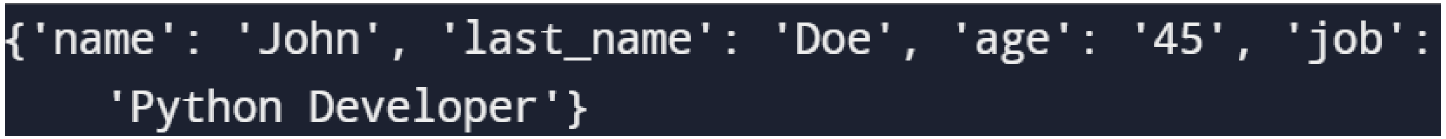
You can also update an existing dictionary by combining zip() with dict.update().

Suppose that John changes his job and you need to update the dictionary. You can do something like the following:

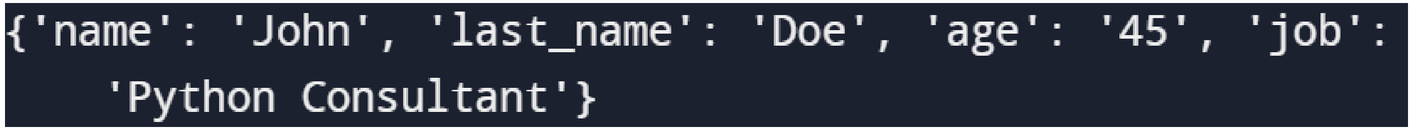
Here, dict.update() updates the dictionary with the key-value tuple you created using Python's zip() function. With this technique, you can easily overwrite the value of job.



```python
fields = ['name', 'last_name', 'age', 'job']
values = ['John', 'Doe', '45', 'Python Developer']

a_dict = dict(zip(fields, values))
print(a_dict)


new_job = ['Python Consultant']
field = ['job']
a_dict.update(zip(field, new_job))
print(a_dict)
```



```
{'name': 'John', 'last_name': 'Doe', 'age': '45', 'job':
    'Python Developer'}
```

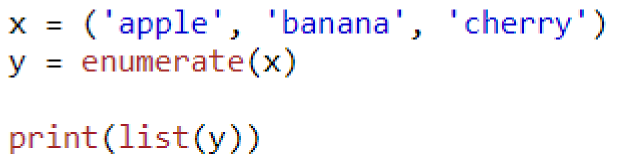


```
{'name': 'John', 'last_name': 'Doe', 'age': '45', 'job':
    'Python Consultant'}
```

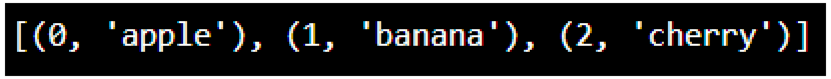
Python Enumerate

- The enumerate() function takes a collection (e.g. a tuple) and returns it as an enumerate object.
- The enumerate() function adds a counter as the key of the enumerate object.
- `enumerate(iterable, start)`
- start : Defines the start number of the enumerate object. *Default 0*
- *Iterable: An Iterable Object*

Enumerate Example



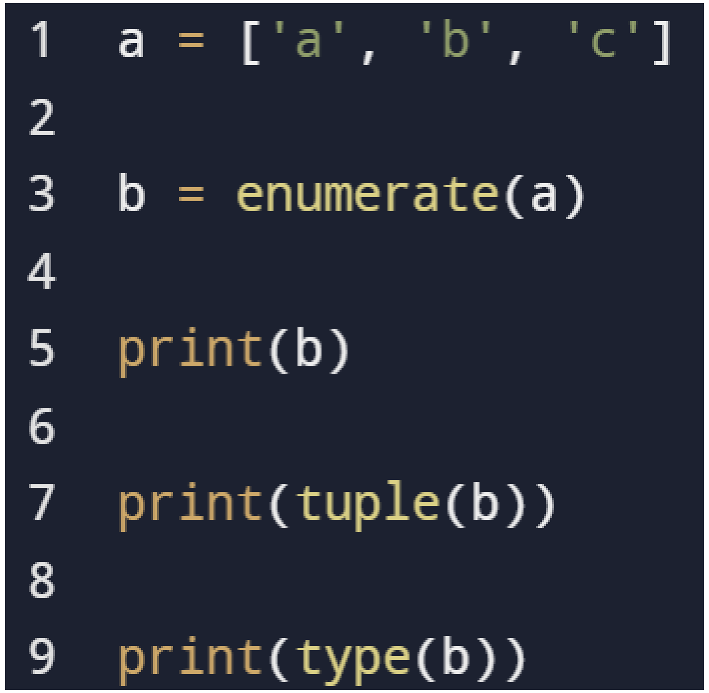
```
x = ('apple', 'banana', 'cherry')
y = enumerate(x)

print(list(y))
```



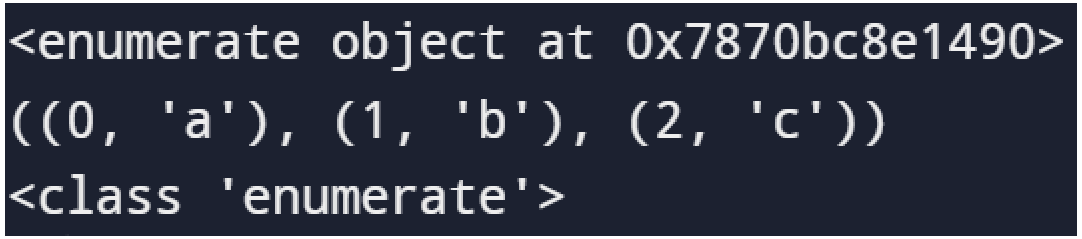
```
[(0, 'apple'), (1, 'banana'), (2, 'cherry')]
```

The enumerate() function in Python takes in a data collection as a parameter and returns an **enumerate object**. The **enumerate object is returned in a key-value pair format**. The key is the corresponding index of each item and the value is the items.

Printing Enumerate



```python
a = ['a', 'b', 'c']

b = enumerate(a)

print(b)

print(tuple(b))

print(type(b))
```

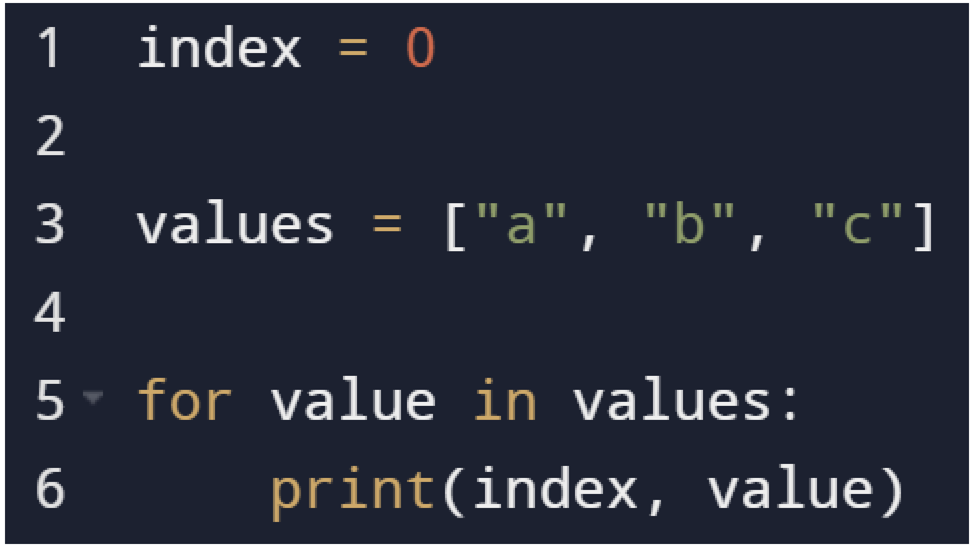


```
<enumerate object at 0x7870bc8e1490>
((0, 'a'), (1, 'b'), (2, 'c'))
<class 'enumerate'>
```

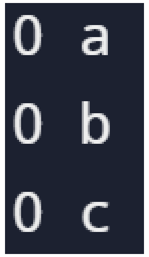
Advantages with Enumerate

Imagine that, in addition to the value itself, you want to print the index of the item in the list to the screen on every iteration.

One way to approach this task is to create a variable to store the index and update it on each iteration.



```python
index = 0

values = ["a", "b", "c"]

for value in values:
    print(index, value)
```

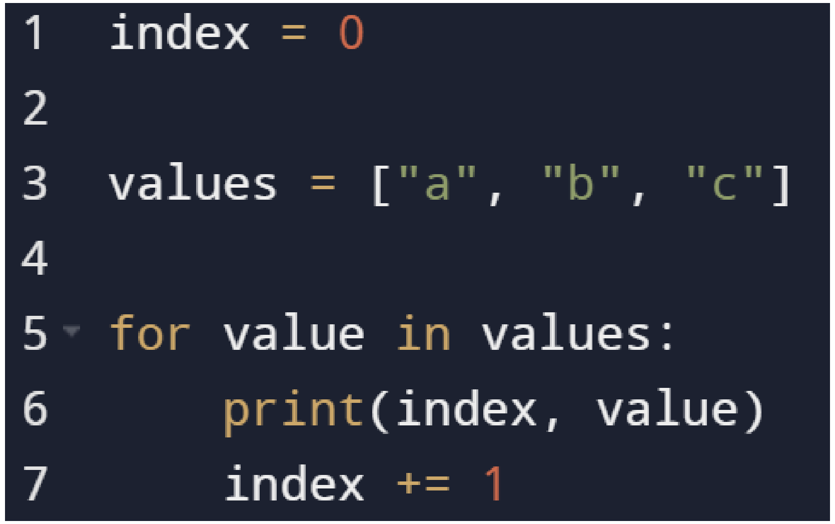


```
0 a
0 b
0 c
```

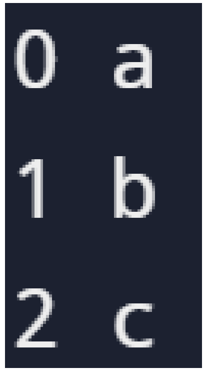
In this example, index stays at 0 on every iteration because there's no code to update its value at the end of the loop. Particularly for long or complicated loops, this kind of bug is notoriously hard to track down.

Advantages with Enumerate

- The usual solution



```python
index = 0

values = ["a", "b", "c"]

for value in values:
    print(index, value)
    index += 1
```

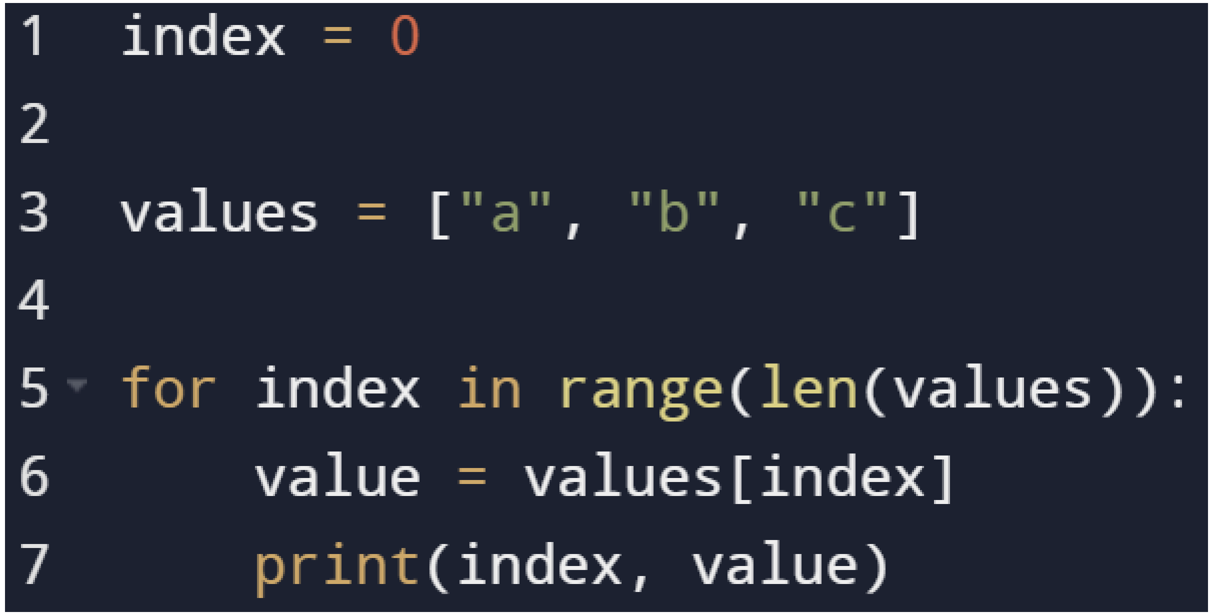


```
0 a
1 b
2 c
```

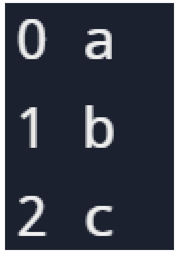
Using range as a solution

In this example, len(values) returns the length of values, which is 3. Then range() creates an iterator running from the default starting value of 0 until it reaches len(values) minus one. In this case, index becomes your loop variable. In the loop, you set value equal to the item in values at the current value of index. Finally, you print index and value.

With this example, one common bug that can occur is when you forget to update value at the beginning of each iteration. This is similar to the previous bug of forgetting to update the index. This is one reason that this loop isn’t considered Pythonic.



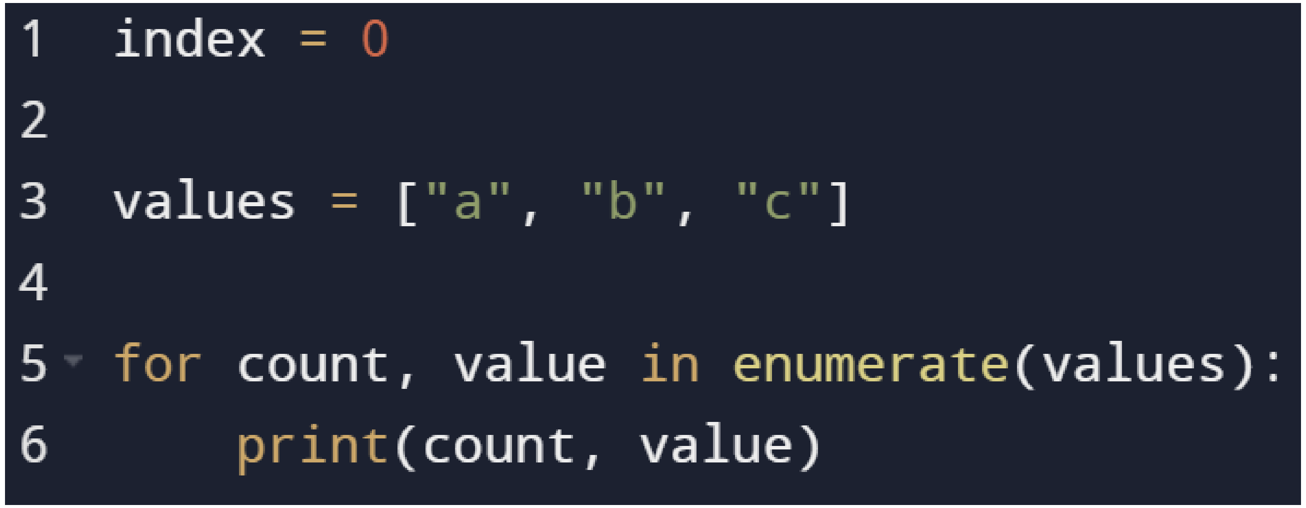
```
index = 0

values = ["a", "b", "c"]

for index in range(len(values)):
    value = values[index]
    print(index, value)
```



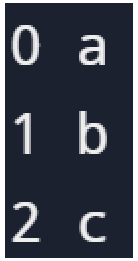
```
0 a
1 b
2 c
```

Working with iterables through Enumerate

You can use enumerate() in a loop in almost the same way that you use the original iterable object. Instead of putting the iterable directly after in in the for loop, you put it inside the parentheses of enumerate(). You also have to change the loop variable a little bit, as shown in this example.



```python
index = 0

values = ["a", "b", "c"]

for count, value in enumerate(values):
    print(count, value)
```



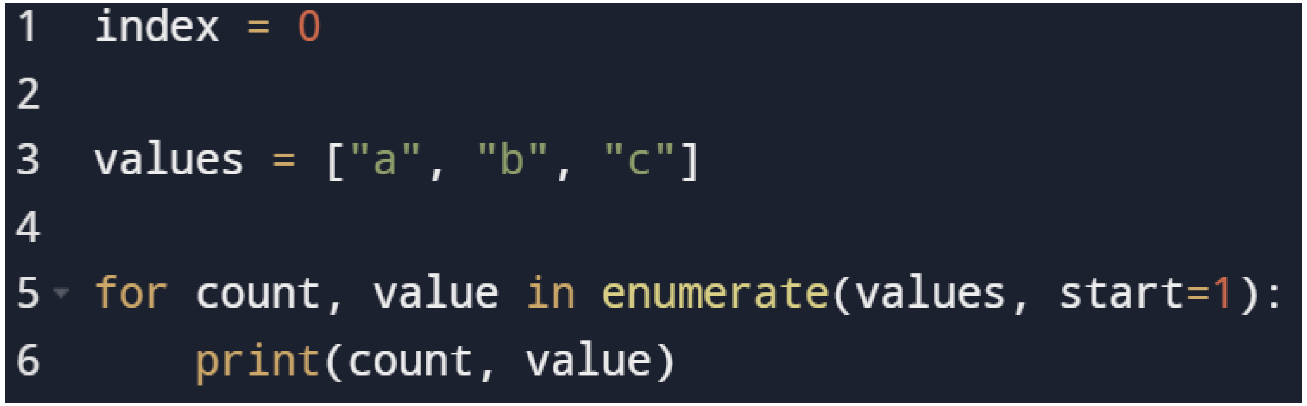
```
0 a
1 b
2 c
```

With enumerate(), you don’t need to remember to access the item from the iterable, and you don’t need to remember to advance the index at the end of the loop.

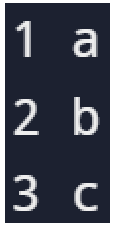
Everything is automatically handled for you by Python!

Changing the start index with Enumerate

Python’s enumerate() has one additional argument that you can use to control the starting value of the count. By default, the starting value is 0 because Python sequence types are indexed starting with zero. In other words, when you want to retrieve the first element of a list, you use index 0:



```python
index = 0

values = ["a", "b", "c"]

for count, value in enumerate(values, start=1):
    print(count, value)
```



```
1 a
2 b
3 c
```

Continuation Next meeting..

Range

Review of Range

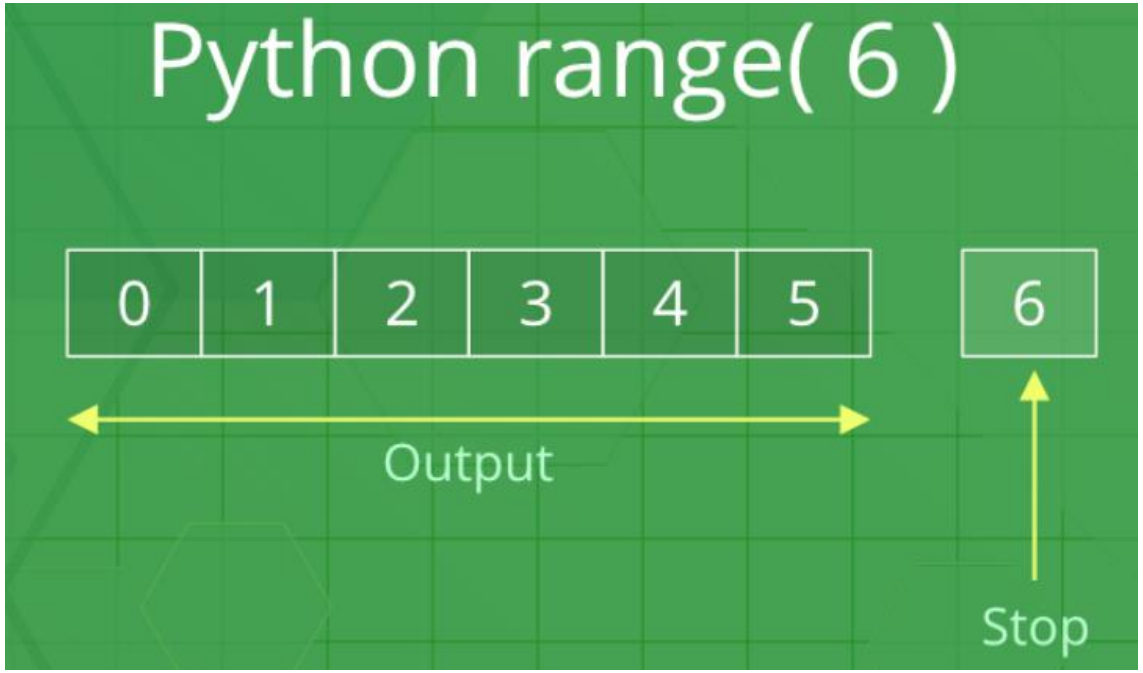
In simple terms, range() allows the user to generate a series of numbers within a given range. Depending on how many arguments the user is passing to the function, the user can decide where that series of numbers will begin and end, as well as how big the difference will be between one number and the next. Python range() function takes can be initialized in 3 ways.

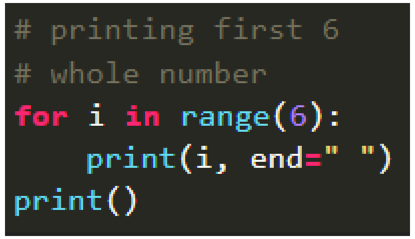
range (stop) takes one argument.

range (start, stop) takes two arguments.

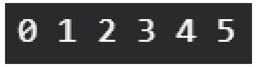
range (start, stop, step) takes three arguments.

When the user call range() with one argument, the user will get a series of numbers that starts at 0 and includes every whole number up to, but not including, the number that the user has provided as the stop.





```
# printing first 6
# whole number
for i in range(6):
    print(i, end=" ")
print()
```

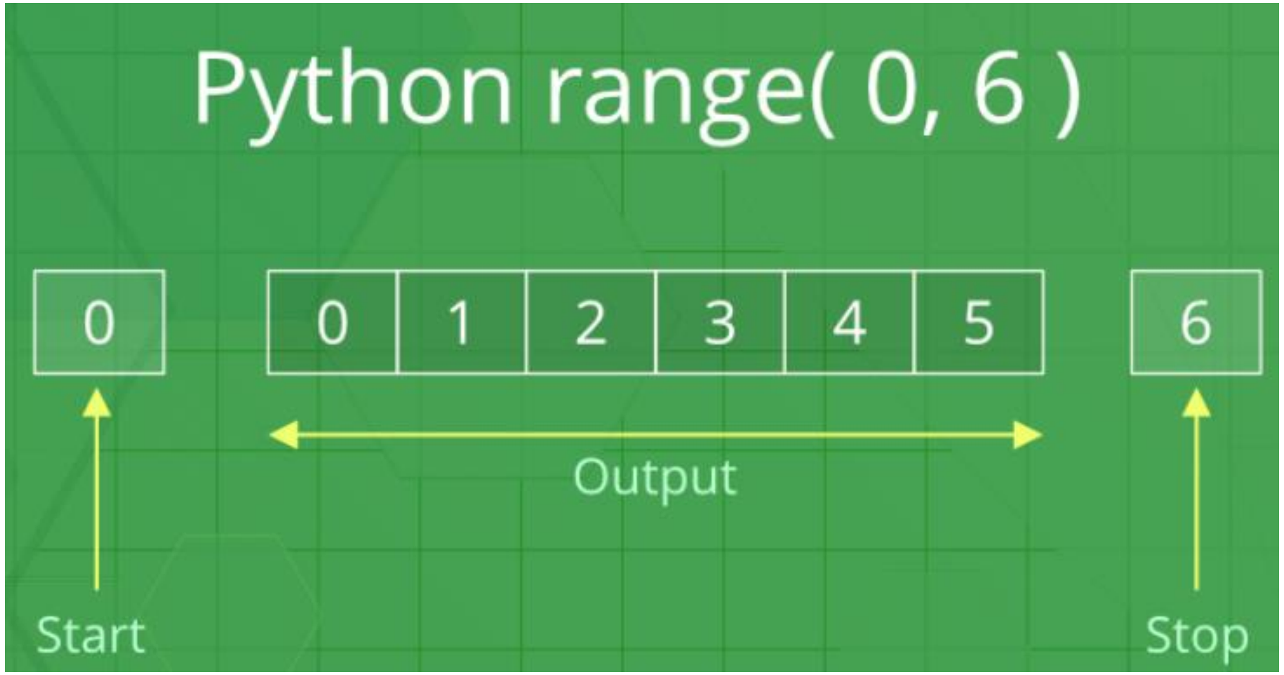


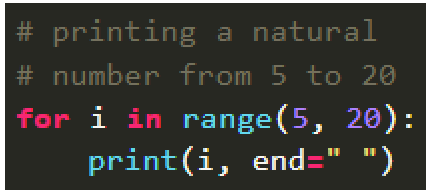
```
0 1 2 3 4 5
```

Range (start, stop)

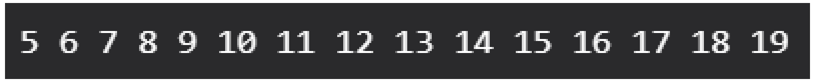
When the user call range() with two arguments, the user gets to decide not only where the series of numbers stops but also where it starts, so the user doesn't have to start at 0 all the time. Users can use range() to generate a series of numbers from X to Y using range(X, Y).

In this example, we are printing the number from 5 to 19. We are using the range function in which we are passing the starting and stopping points of the loop.





```
# printing a natural
# number from 5 to 20
for i in range(5, 20):
    print(i, end=" ")
```

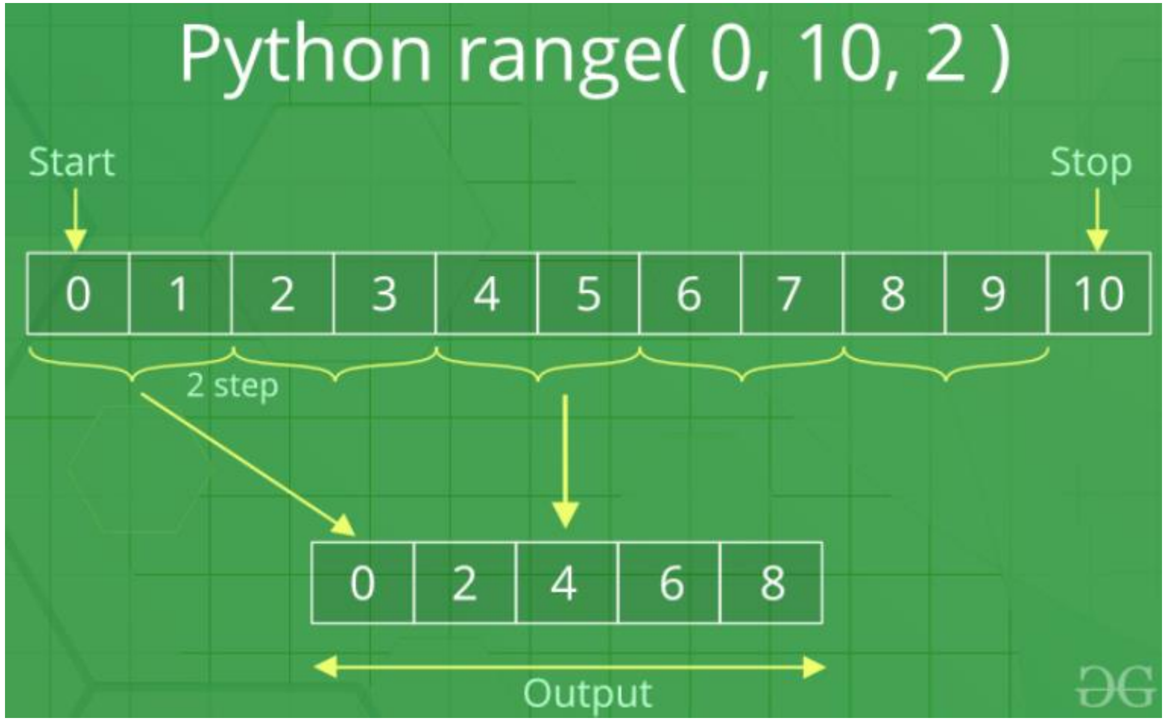


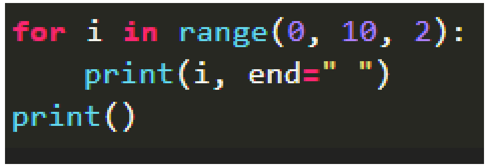
```
5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

Python range (start, stop, step)

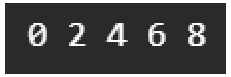
Python range (start, stop, step)

When the user call range() with three arguments, the user can choose not only where the series of numbers will start and stop, but also how big the difference will be between one number and the next. If the user doesn't provide a step, then range() will automatically behave as if the step is 1. In this example, we are printing even numbers between 0 and 10, so we choose our starting point from 0(start = 0) and stop the series at 10(stop = 10). For printing an even number the difference between one number and the next must be 2 (step = 2) after providing a step we get the following output (0, 2, 4, 8).



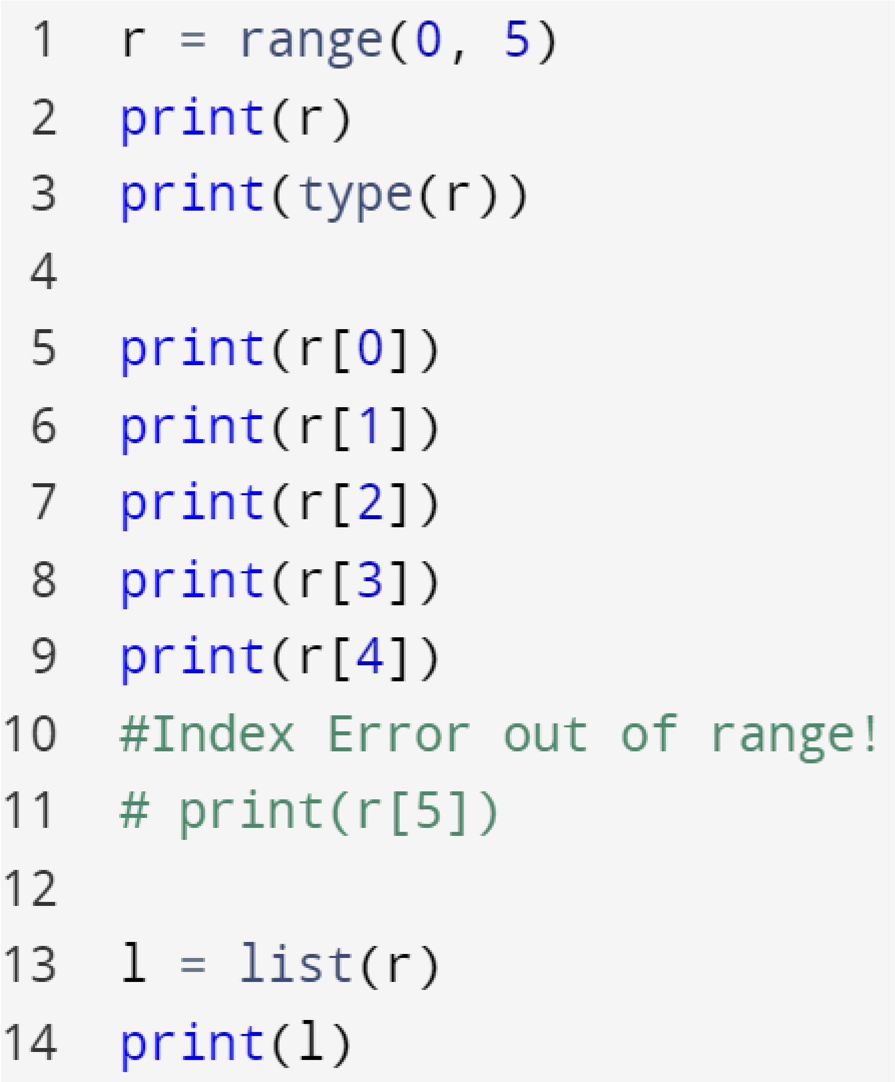


```
for i in range(0, 10, 2):
    print(i, end=" ")
print()
```

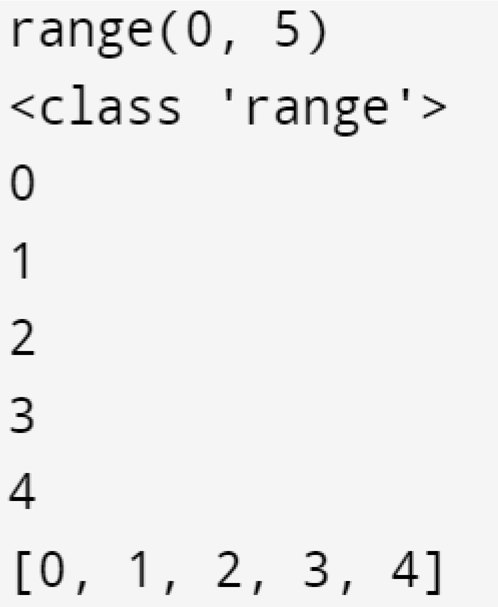


```
0 2 4 6 8
```

Printing Range



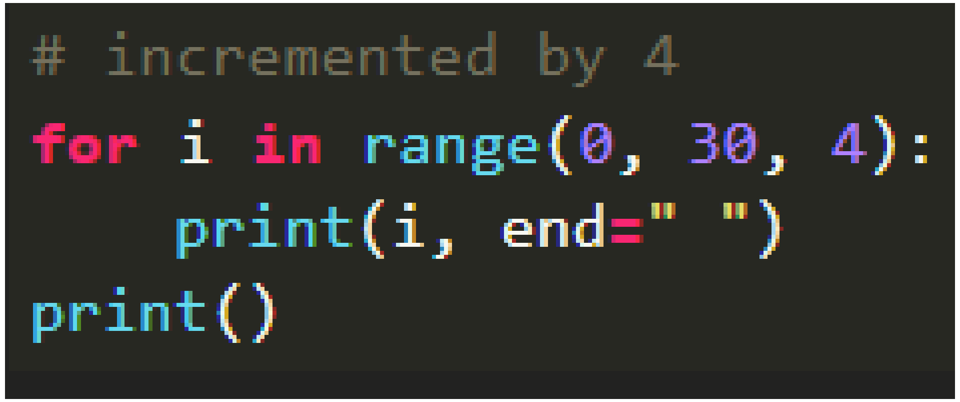
```python
r = range(0, 5)
print(r)
print(type(r))

print(r[0])
print(r[1])
print(r[2])
print(r[3])
print(r[4])
#Index Error out of range!
# print(r[5])

l = list(r)
print(l)
```



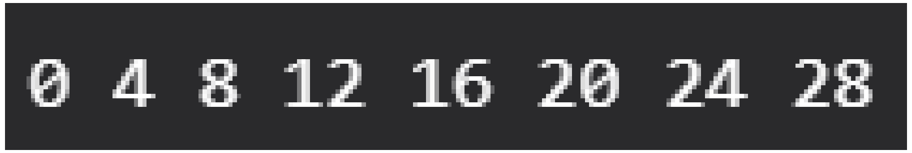
```
range(0, 5)
<class 'range'>
0
1
2
3
4
[0, 1, 2, 3, 4]
```

Incrementing the Range using a Positive Step

- If a user wants to increment, then the user needs steps to be a positive number.



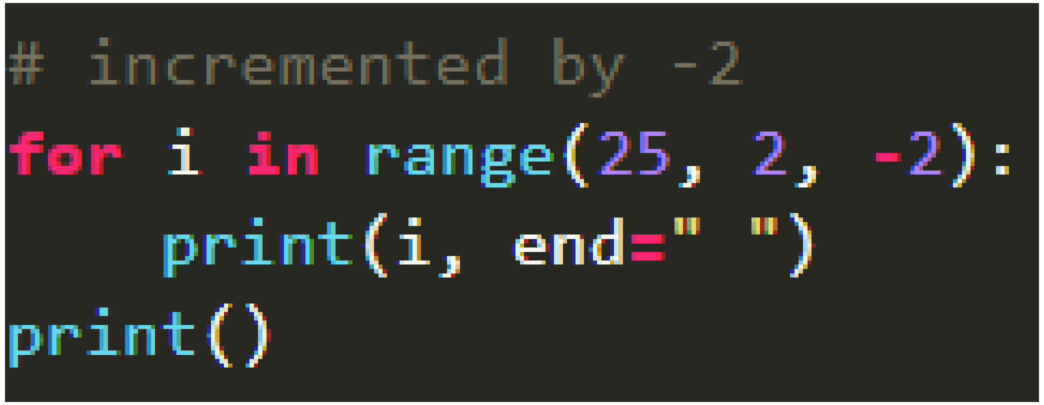
```python
# incremented by 4
for i in range(0, 30, 4):
    print(i, end=" ")
print()
```



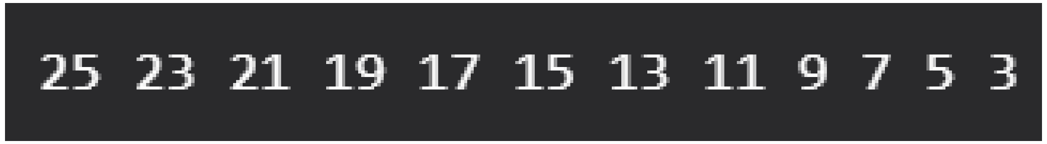
```
0 4 8 12 16 20 24 28
```

Python range() using Negative Step

- If a user wants to decrement, then the user needs steps to be a negative number.



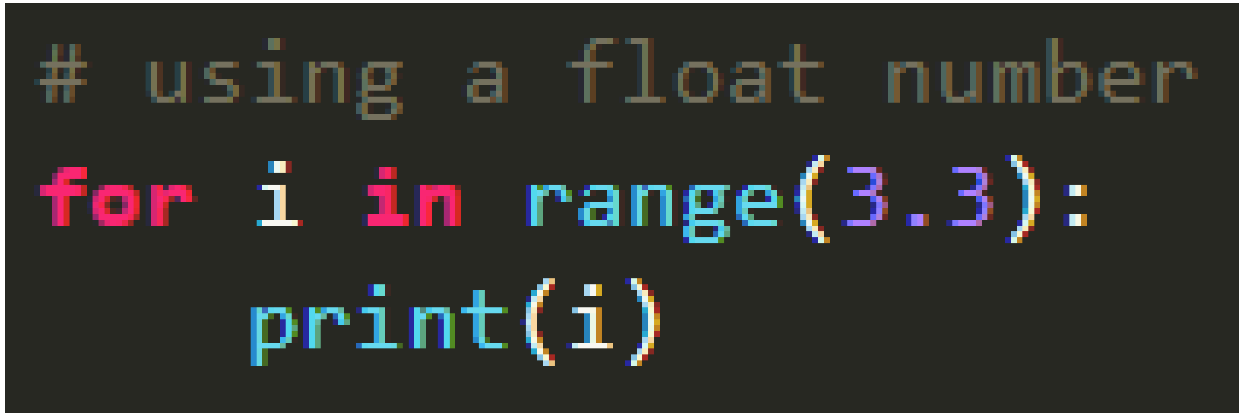
```
# incremented by -2
for i in range(25, 2, -2):
    print(i, end=" ")
print()
```



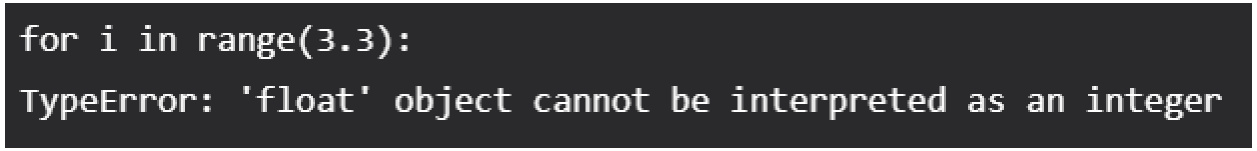
```
25 23 21 19 17 15 13 11 9 7 5 3
```

Python range() with Float Values

- Python range() function doesn’t support float numbers. i.e. user cannot use floating-point or non-integer numbers in any of its arguments. Users can use only integer numbers.



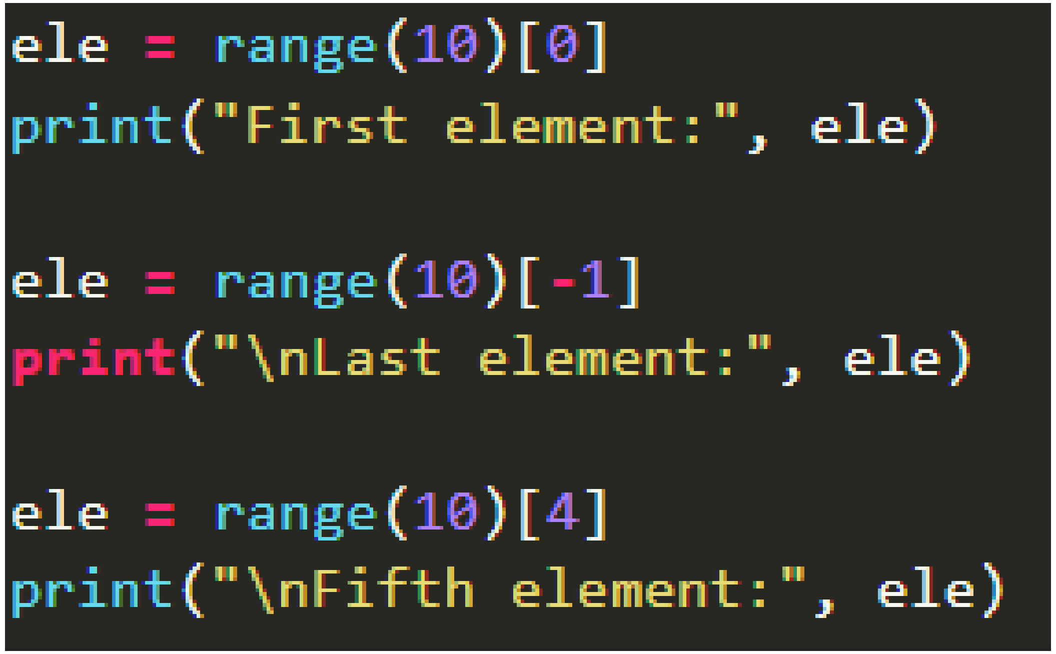
```
# using a float number
for i in range(3.3):
    print(i)
```



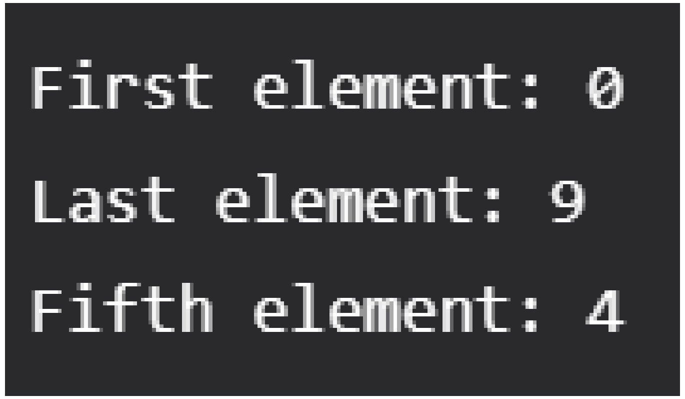
```
for i in range(3.3):
TypeError: 'float' object cannot be interpreted as an integer
```

Accessing range() with an Index Value

- A sequence of numbers is returned by the range() function as its object that can be accessed by its index value. Both positive and negative indexing is supported by its object.



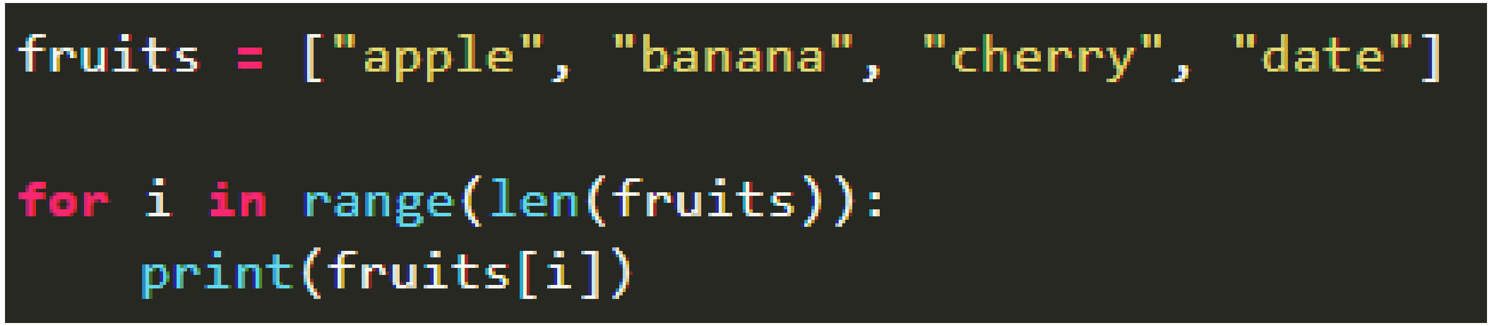
```python
ele = range(10)[0]
print("First element:", ele)

ele = range(10)[-1]
print("\nLast element:", ele)

ele = range(10)[4]
print("\nFifth element:", ele)
```



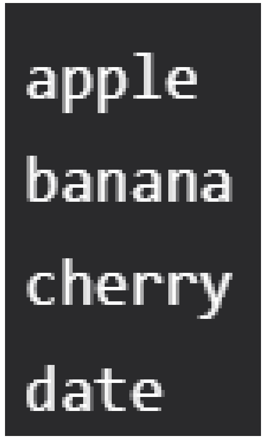
```
First element: 0

Last element: 9

Fifth element: 4
```

range() function with List in Python

- In this example, we are creating a list and we are printing list elements with the range() in Python.



```
fruits = ["apple", "banana", "cherry", "date"]

for i in range(len(fruits)):
    print(fruits[i])
```



```
apple
banana
cherry
date
```

Key things to remember with range

- The range() function only works with integers, i.e. whole numbers.
- All arguments must be integers. Users can not pass a string or float number or any other type in a start, stop, and step argument of a range().
- All three arguments can be positive or negative.
- The step value must not be zero. If a step is zero, python raises a ValueError exception.
- Users can access items in a range() by index, just as users do with a list.

Nested if

Nested If

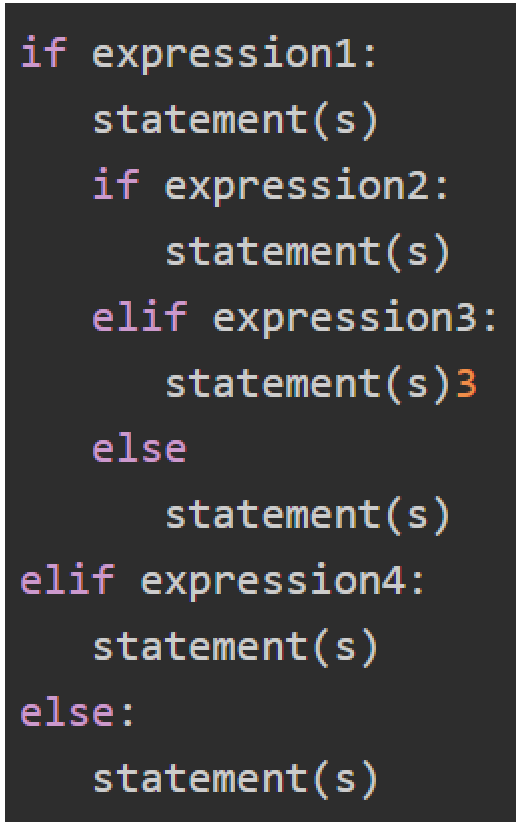
- Python supports nested if statements which means we can use a conditional if of else...if statement inside an existing if statement.

- There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested if construct.

- In a nested if construct, you can have an if...elif...else construct inside another if...elif...else construct.

Nested if Syntax

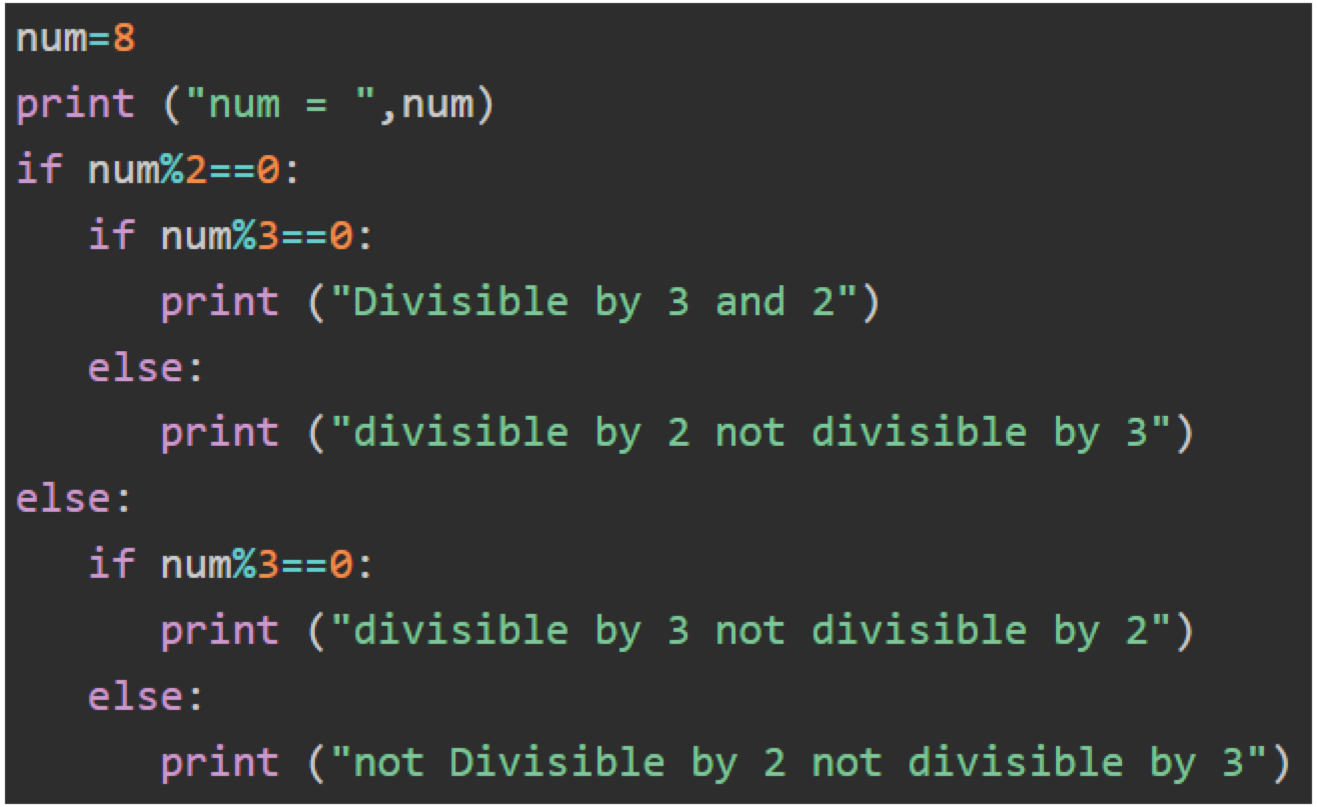
- The syntax of the nested if...elif...else construct will be like this –



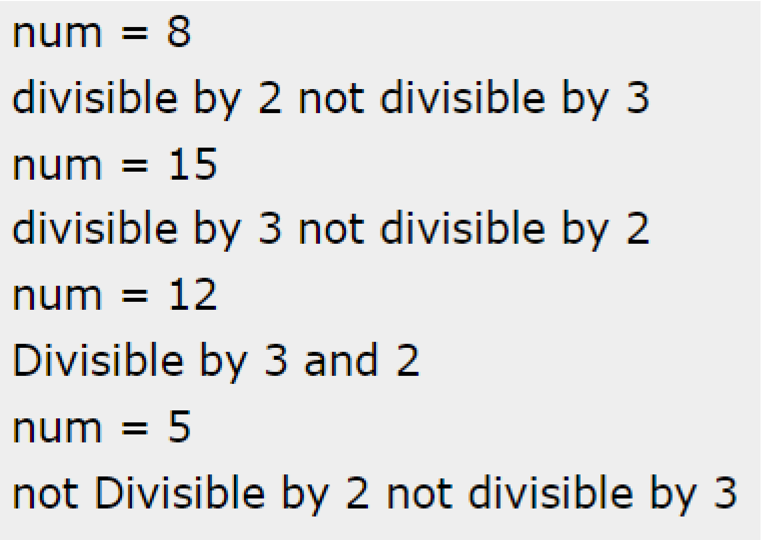
```
if expression1:
   statement(s)
   if expression2:
      statement(s)
   elif expression3:
      statement(s)3
   else
      statement(s)
elif expression4:
   statement(s)
else:
   statement(s)
```

Nested if Example

- Now let's take a Python code to understand how it works –



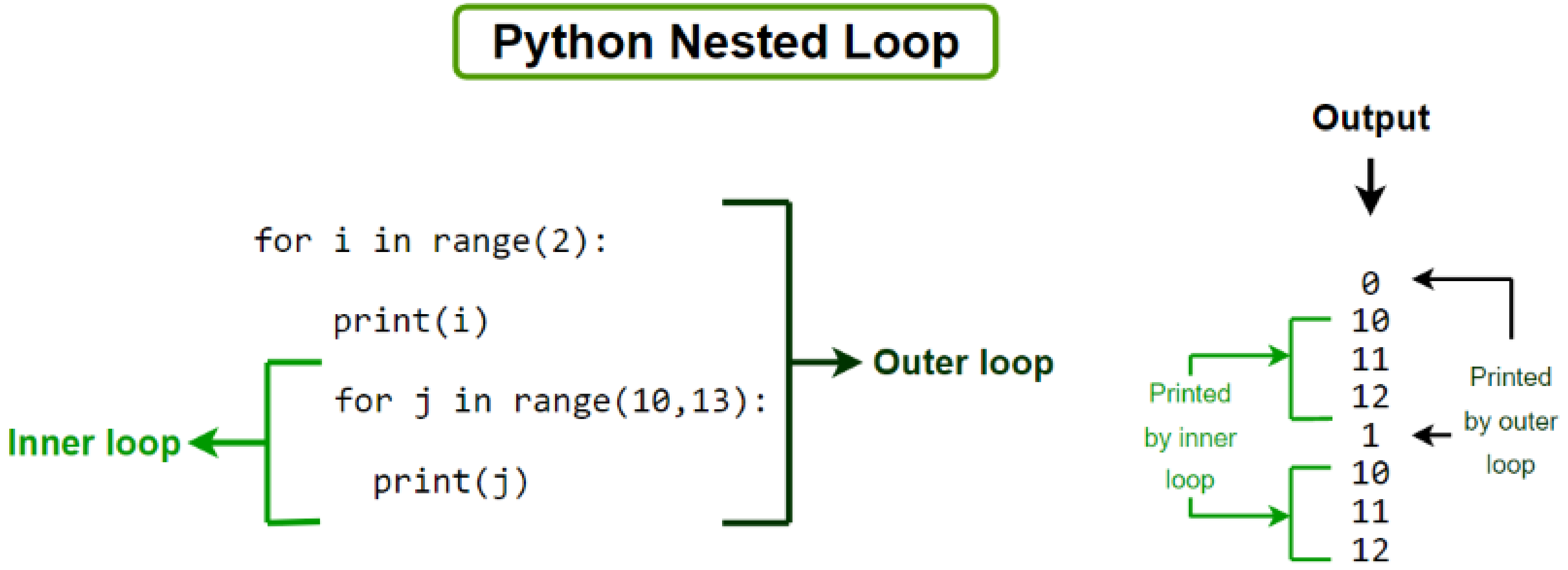
```
num=8
print ("num = ",num)
if num%2==0:
   if num%3==0:
      print ("Divisible by 3 and 2")
   else:
      print ("divisible by 2 not divisible by 3")
else:
   if num%3==0:
      print ("divisible by 3 not divisible by 2")
   else:
      print ("not Divisible by 2 not divisible by 3")
```



```
num = 8
divisible by 2 not divisible by 3
num = 15
divisible by 3 not divisible by 2
num = 12
Divisible by 3 and 2
num = 5
not Divisible by 2 not divisible by 3
```

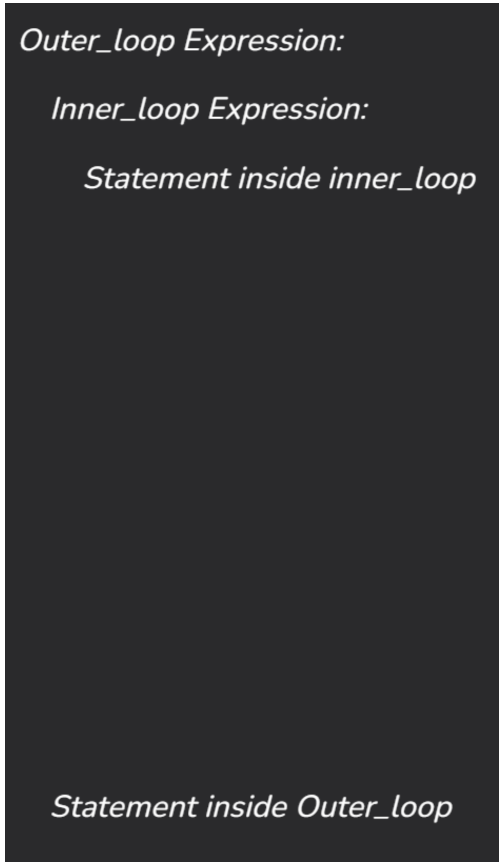
Nested Loops

Nested Loops



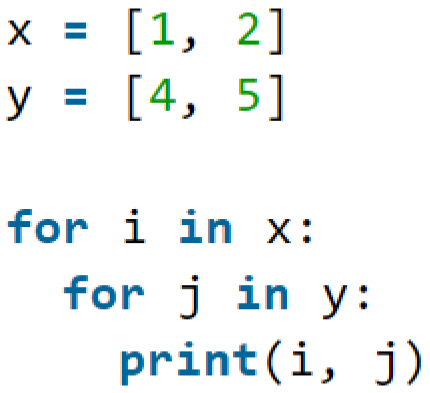
```
for i in range(2):
    print(i)
    for j in range(10,13):
      print(j)
```

```
0
10
11
12
1
10
11
12
```

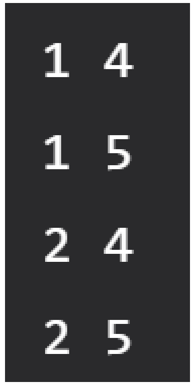


```
Outer_loop Expression:
  Inner_loop Expression:
    Statement inside inner_loop

  Statement inside Outer_loop
```

Python Nested Loops Examples

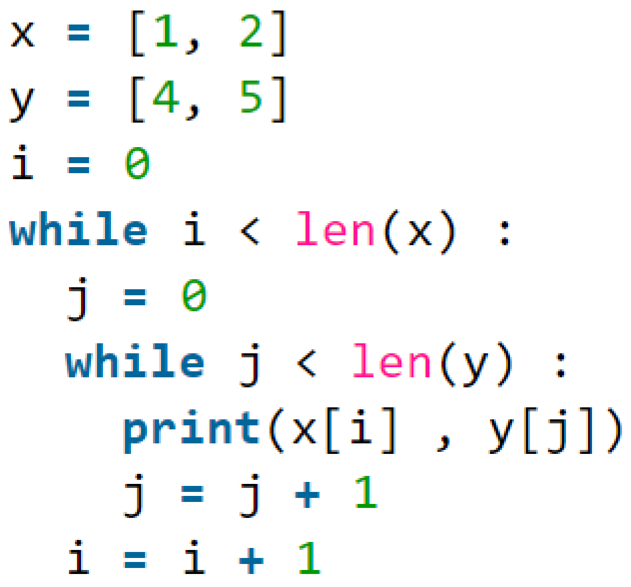


```
x = [1, 2]
y = [4, 5]

for i in x:
  for j in y:
    print(i, j)
```

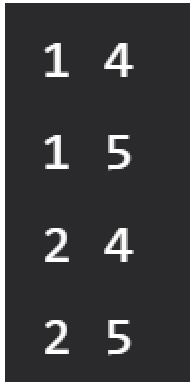


```
1 4
1 5
2 4
2 5
```

Python Nested Loops Examples



```
x = [1, 2]
y = [4, 5]
i = 0
while i < len(x) :
  j = 0
  while j < len(y) :
    print(x[i] , y[j])
    j = j + 1
  i = i + 1
```



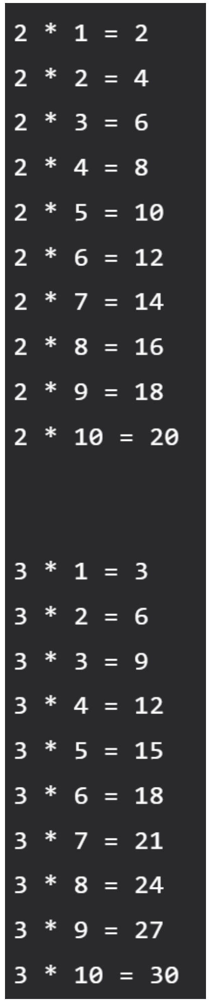
```
1 4
1 5
2 4
2 5
```

Printing multiplication table using Python nested for loops

In this example what we do is take an outer for loop running from 2 to 3 for multiplication table of 2 and 3 and then inside that loop we are taking an inner for loop that will run from 1 to 10 inside that we are printing multiplication table by multiplying each iteration value of inner loop with the iteration value of outer loop as we see in the below output.

```
# Running outer loop from 2 to 3

for i in range(2, 4):

    # Printing inside the outer loop
    # Running inner loop from 1 to 10
    for j in range(1, 11):

        # Printing inside the inner loop
        print(i, "*", j, "=", i*j)
    # Printing inside the outer loop
    print()
```

×10

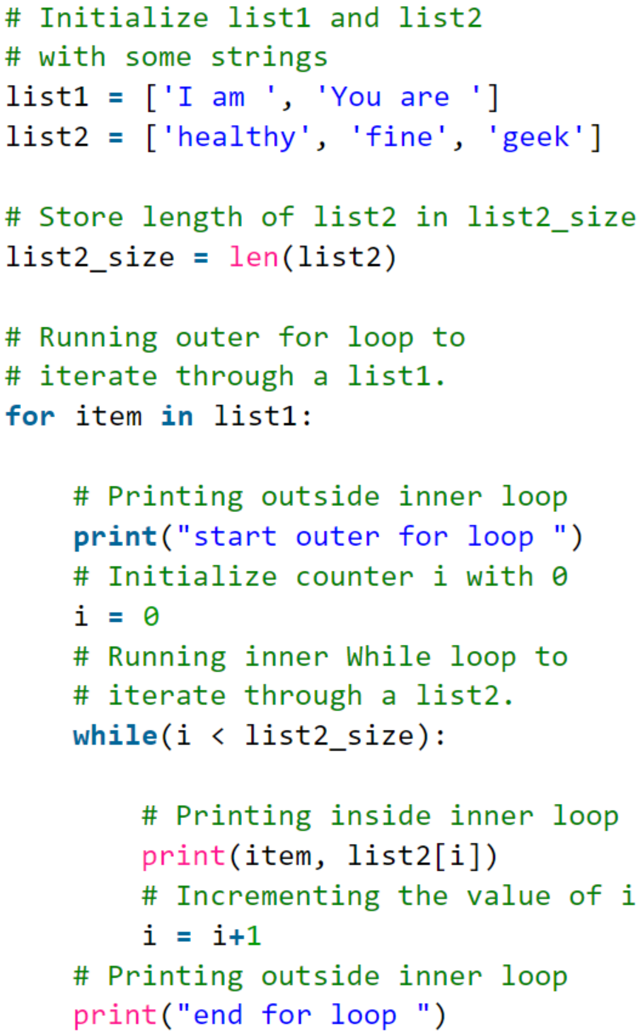


```
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
2 * 10 = 20


3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
3 * 4 = 12
3 * 5 = 15
3 * 6 = 18
3 * 7 = 21
3 * 8 = 24
3 * 9 = 27
3 * 10 = 30
```

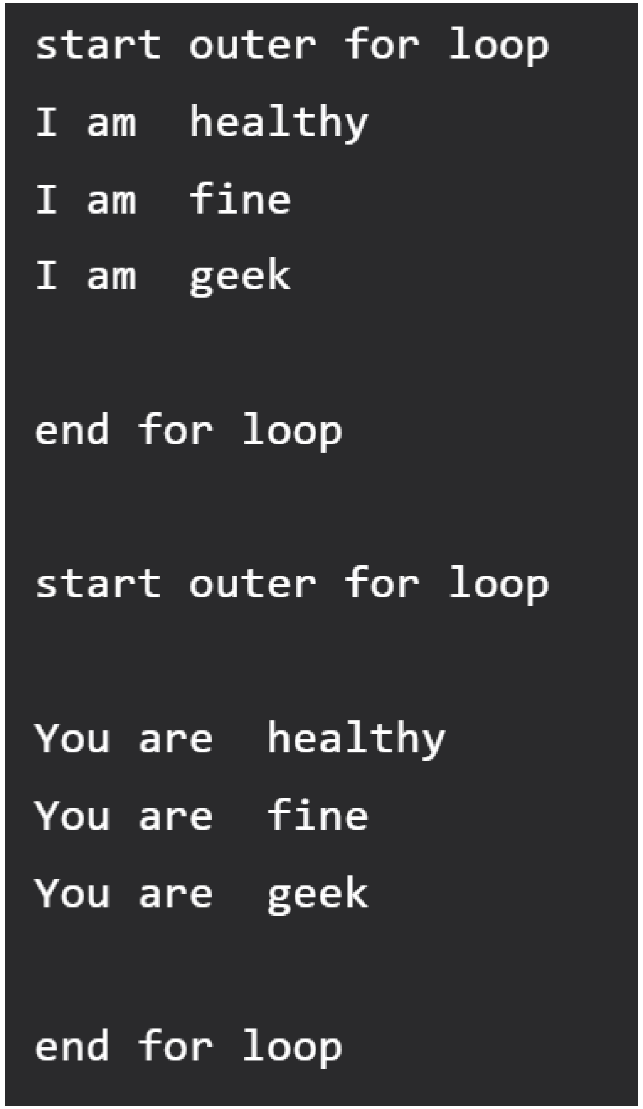
Printing using different inner and outer nested loops

In this example, we are initializing two lists with some strings. Store the size of list2 in ‘list2_Size’ using len() function and using it in the while loop as a counter.

After that run, an outer for loop to iterate over list1 and inside that loop run an inner while loop to iterate over list2 using list indexing inside that we are printing each value of list2 for every value of list1.



```python
# Initialize list1 and list2
# with some strings
list1 = ['I am ', 'You are ']
list2 = ['healthy', 'fine', 'geek']

# Store length of list2 in list2_size
list2_size = len(list2)

# Running outer for loop to
# iterate through a list1.
for item in list1:

    # Printing outside inner loop
    print("start outer for loop ")
    # Initialize counter i with 0
    i = 0
    # Running inner While loop to
    # iterate through a list2.
    while(i < list2_size):

        # Printing inside inner loop
        print(item, list2[i])
        # Incrementing the value of i
        i = i+1
    # Printing outside inner loop
    print("end for loop ")
```



```
start outer for loop
I am  healthy
I am  fine
I am  geek

end for loop

start outer for loop

You are  healthy
You are  fine
You are  geek

end for loop
```

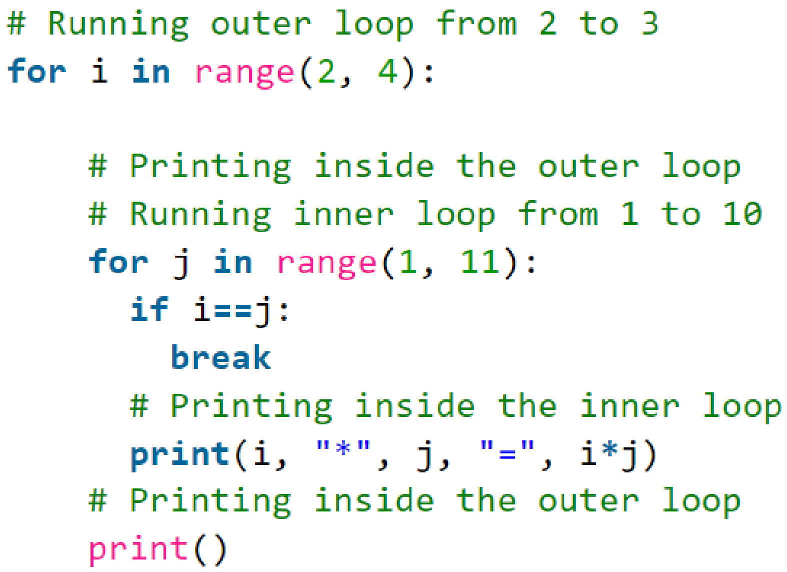
Using break statement in nested loops

It is a type of loop control statement. In a loop, we can use the break statement to exit from the loop.

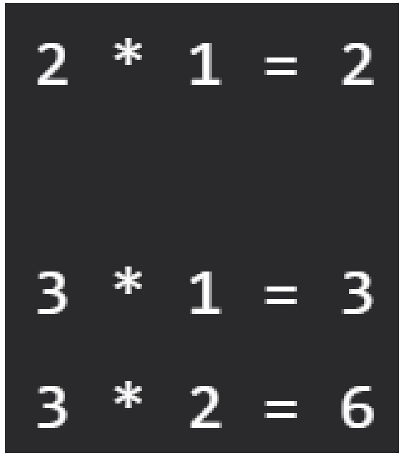
When we use a break statement in a loop it skips the rest of the iteration and terminates the loop. let’s understand it using an example.

In this code we are using a break statement inside the inner loop by using the if statement.

Inside the inner loop if ‘i’ becomes equals to ‘j’ then the inner loop will be terminated and not executed the rest of the iteration as we can see in the output table of 3 is printed up to two iterations because in the next iteration ‘i’ becomes equal to ‘j’ and the loop breaks.



```
# Running outer loop from 2 to 3
for i in range(2, 4):

  # Printing inside the outer loop
  # Running inner loop from 1 to 10
  for j in range(1, 11):
   if i==j:
    break
   # Printing inside the inner loop
   print(i, "*", j, "=", i*j)
  # Printing inside the outer loop
  print()
```



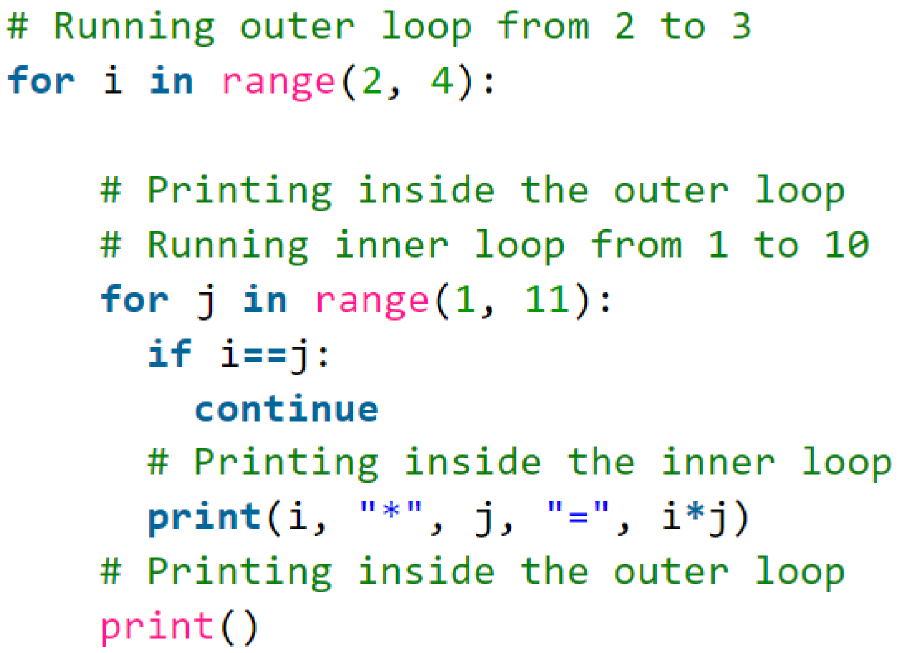
```
2 * 1 = 2

3 * 1 = 3
3 * 2 = 6
```

Using continue statement in nested loops

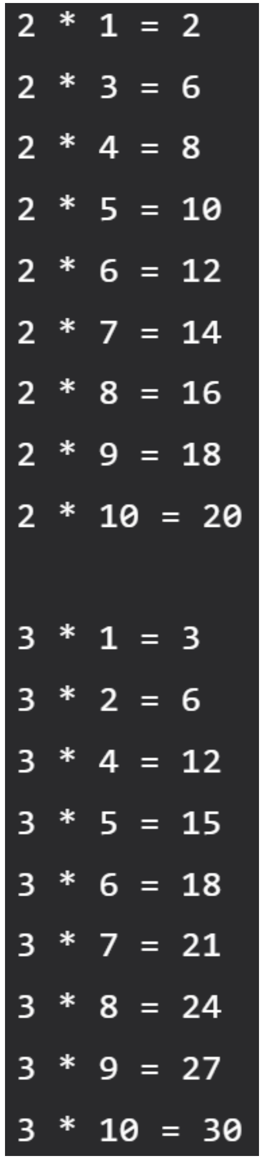
A continue statement is also a type of loop control statement. It is just the opposite of the break statement.

The continue statement forces the loop to jump to the next iteration of the loop whereas the break statement terminates the loop. Let’s understand it by using code.

Instead of using a break statement, we are using a continue statement. Here when ‘i’ becomes equal to ‘j’ in the inner loop it skips the rest of the code in the inner loop and jumps on the next iteration as we see in the output “2 * 2 = 4” and “3 * 3 = 9” is not printed because at that point ‘i’ becomes equal to ‘j’.



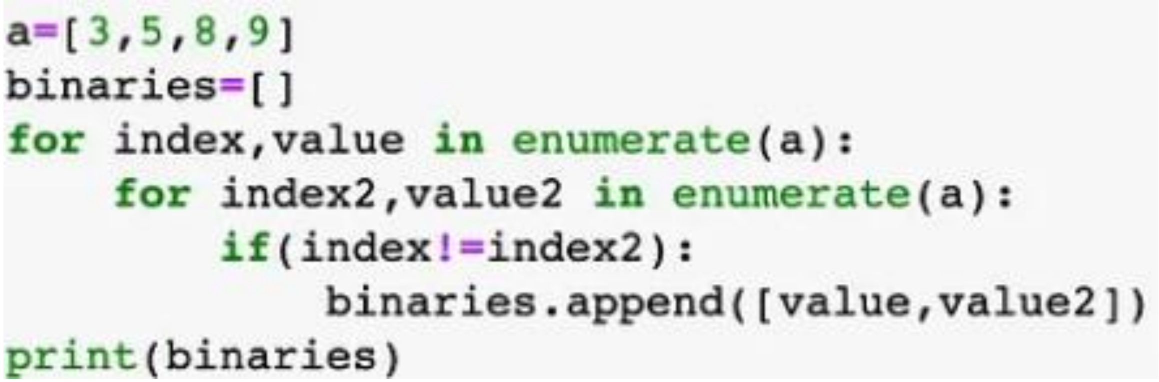
```
# Running outer loop from 2 to 3
for i in range(2, 4):

    # Printing inside the outer loop
    # Running inner loop from 1 to 10
    for j in range(1, 11):
      if i==j:
        continue
      # Printing inside the inner loop
      print(i, "*", j, "=", i*j)
    # Printing inside the outer loop
    print()
```



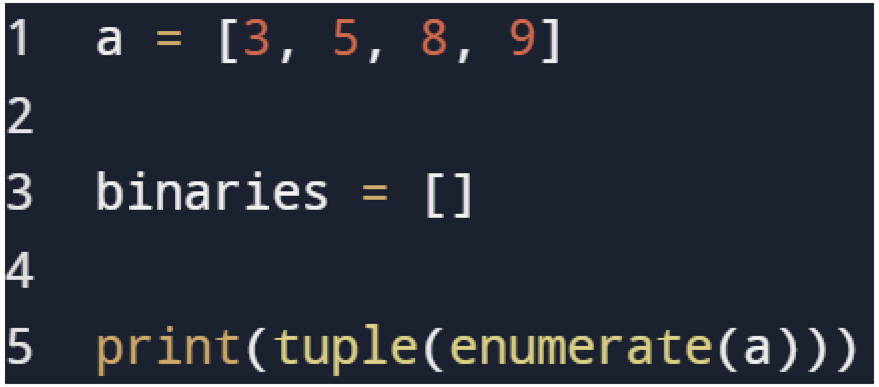
```
2 * 1 = 2
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
2 * 10 = 20

3 * 1 = 3
3 * 2 = 6
3 * 4 = 12
3 * 5 = 15
3 * 6 = 18
3 * 7 = 21
3 * 8 = 24
3 * 9 = 27
3 * 10 = 30
```

Nested Loops with Enumerate and Methods

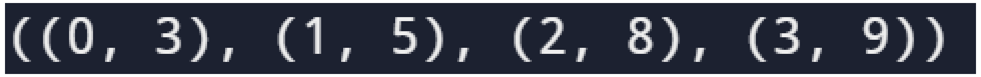
Nested For Loop Using Enumerate()



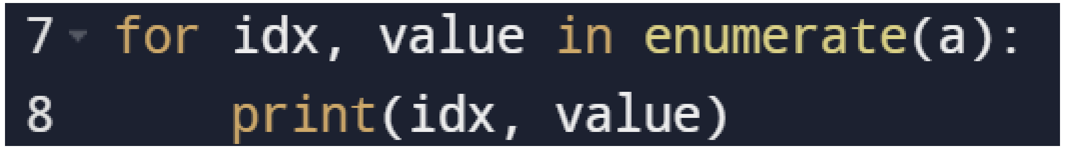
```
a=[3,5,8,9]
binaries=[]
for index,value in enumerate(a):
    for index2,value2 in enumerate(a):
        if(index!=index2):
            binaries.append([value,value2])
print(binaries)
```



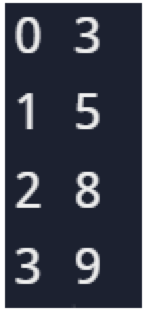
```
a = [3, 5, 8, 9]

binaries = []

print(tuple(enumerate(a)))
```



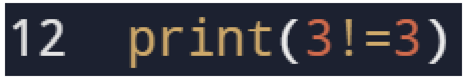
```
((0, 3), (1, 5), (2, 8), (3, 9))
```



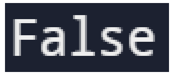
```
for idx, value in enumerate(a):
    print(idx, value)
```



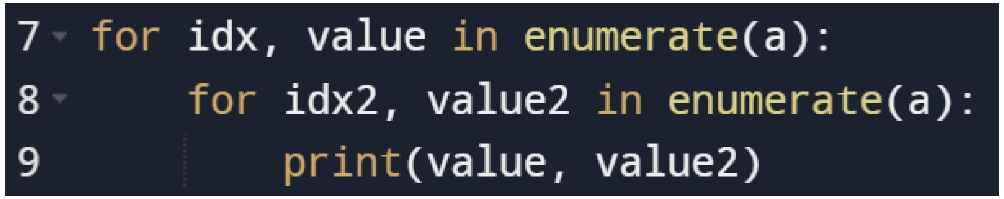
```
0 3
1 5
2 8
3 9
```



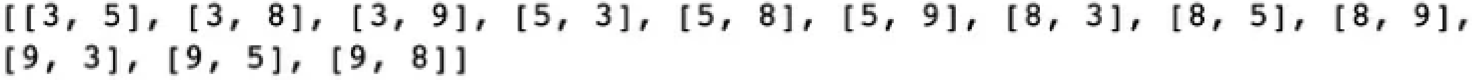
```
print(3!=3)
```



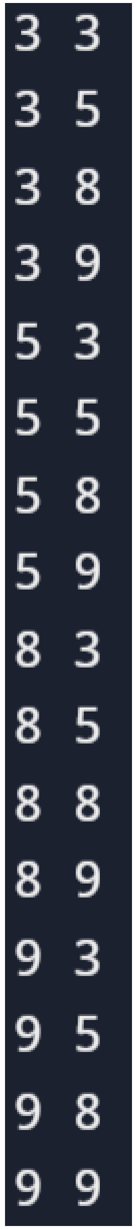
```
False
```



```
for idx, value in enumerate(a):
    for idx2, value2 in enumerate(a):
        print(value, value2)
```



```
[[3, 5], [3, 8], [3, 9], [5, 3], [5, 8], [5, 9], [8, 3], [8, 5], [8, 9],
[9, 3], [9, 5], [9, 8]]
```



```
3 3
3 5
3 8
3 9
5 3
5 5
5 8
5 9
8 3
8 5
8 8
8 9
9 3
9 5
9 8
9 9
```