Week 3

#Data Science/4 - Python for Data Science, AI & Development#

Conditions and Branching

In this video, you will learn about **conditions** and **branching**.

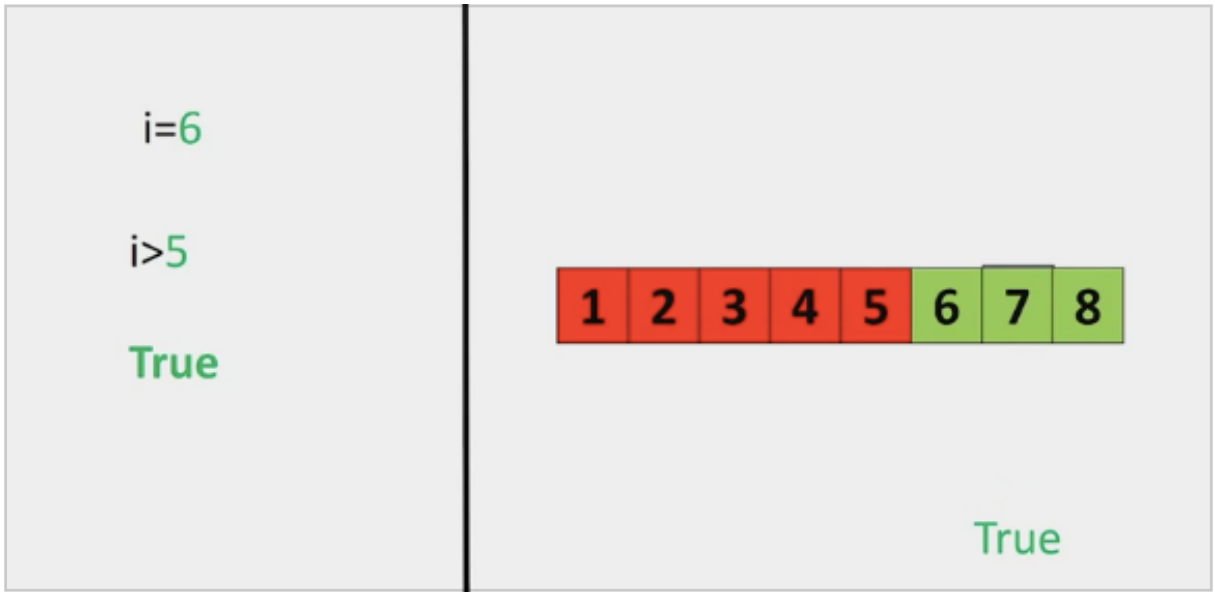
Comparison

Comparison operations compares some value or operand. Then based on some condition, they produce a Boolean.

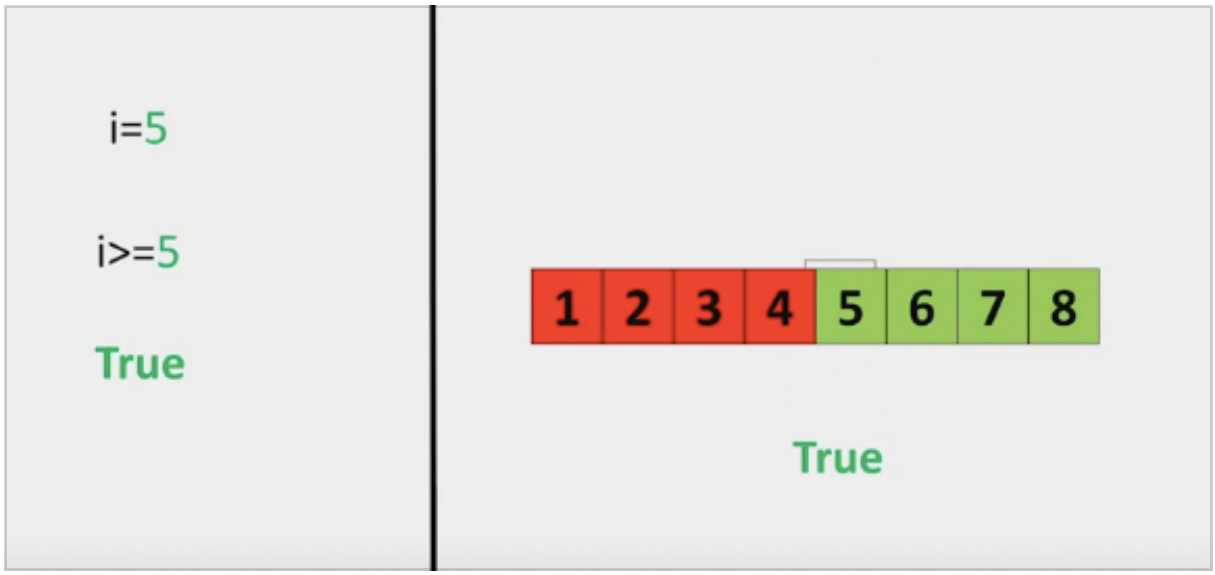
Let's say we assign a value of a to six. We can use the equality operator denoted with two equal signs to determine if two values are equal. In this case, if seven is equal to six. In this case, as six is not equal to seven, the result is false. If we performed an equality test for the value six, the two values would be equal. As a result, we would get a true.

Consider the following equality comparison operator: If the value of the left operand, in this case, the variable I is greater than the value of the right operand, in this case five, the condition becomes true or else we get a false.

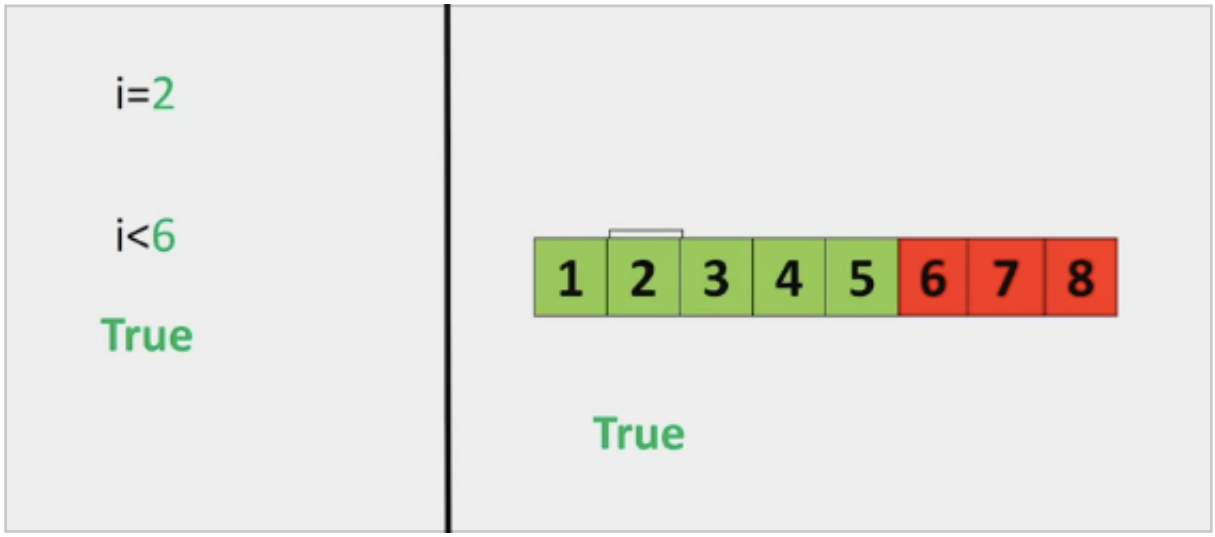
Let's display some values for I on the left. Let's see the value is greater than five in green and the rest in red. If we set I equal to six, we see that six is larger than five and as a result, we get a true. We can also apply the same operations to floats.



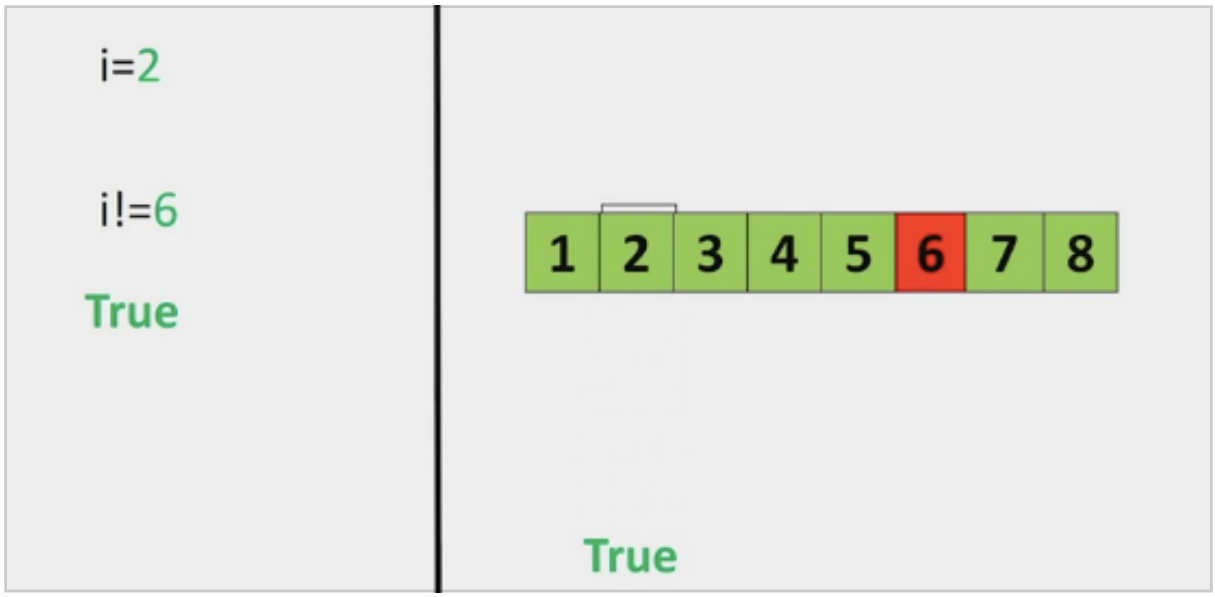
If we modify the operator as follows, if the left operand I is greater than or equal to the value of the right operand, in this case five, then the condition becomes true. In this case, we include the value of five in the number line and the color changes to green accordingly. If we set the value of I equal to five, the operand will produce a true.



If we set the value of I to two, we would get a false because two is less than five.
We can change the inequality if the value of the left operand, in this case, I is less than the value of the right operand, in this case, six. Then condition becomes true. Again, we can represent this with a colored number line. The areas where the inequality is true are marked in green and red where the inequality is false.

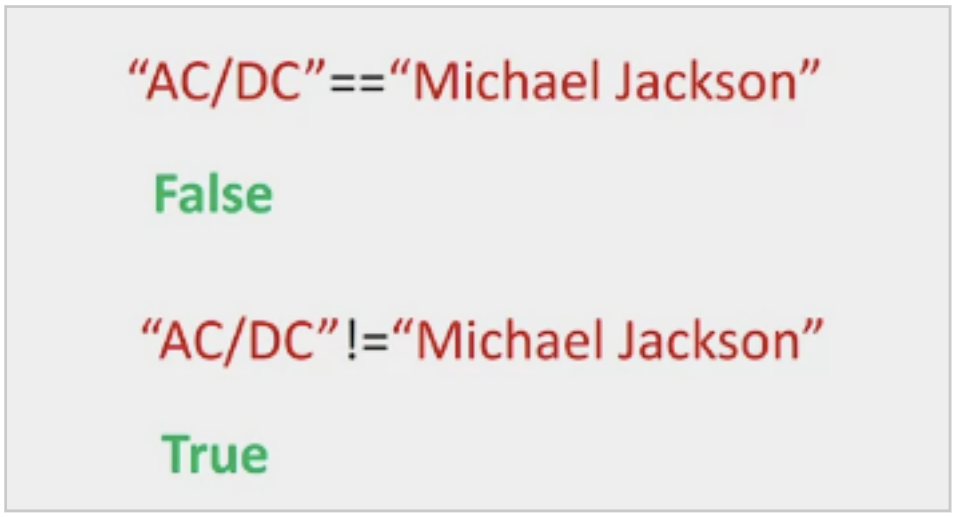


If the value for I is set to two, the result is a true. As two is less than six.
The **inequality** test **uses an explanation mark preceding the equal sign** `!=`. If two operands are not equal, then the condition becomes true. We can use a number line. When the condition is true, the corresponding numbers are marked in green and red for where the condition is false.



If we set I equal to two, the operator is true as two is not equal to six.

We compare strings as well. Comparing ACDC and Michael Jackson using the equality test, we get a false, as the strings are not the same. Using the inequality test, we get a true, as the strings are different.



See the LAB for more examples.

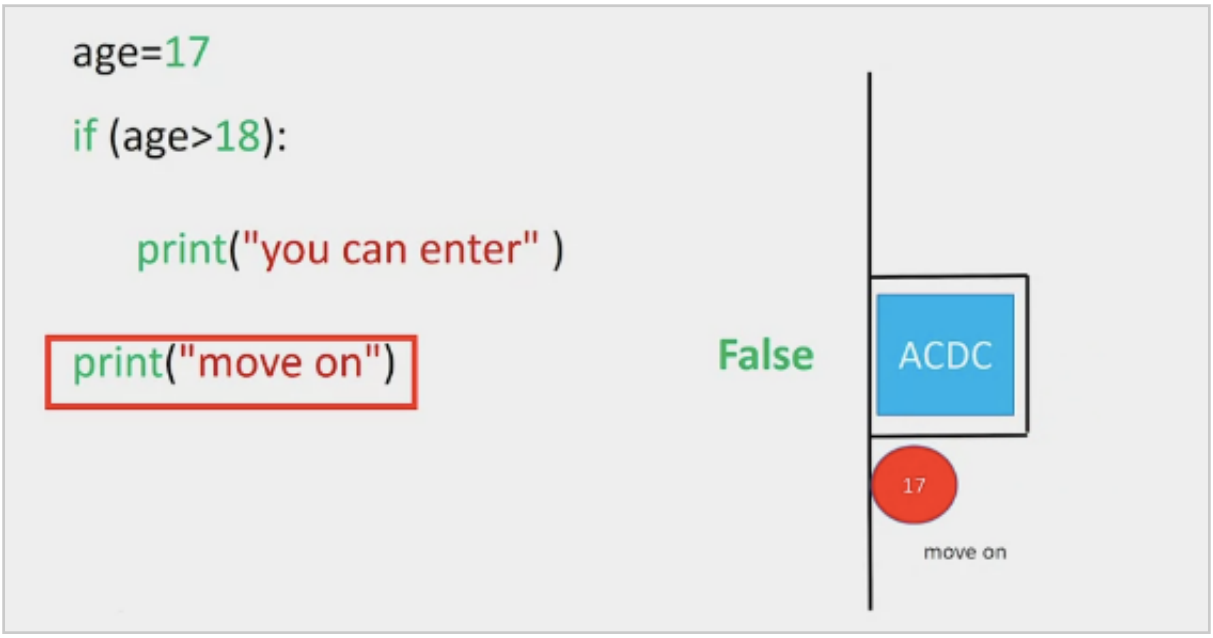
Branching theory

Branching allows us to run different statements for a different input. It's helpful to think of an `if` statement as a locked room. If this statement is true, you can enter the room and your program can run some predefined task. If the statement is false, your program will skip the task.

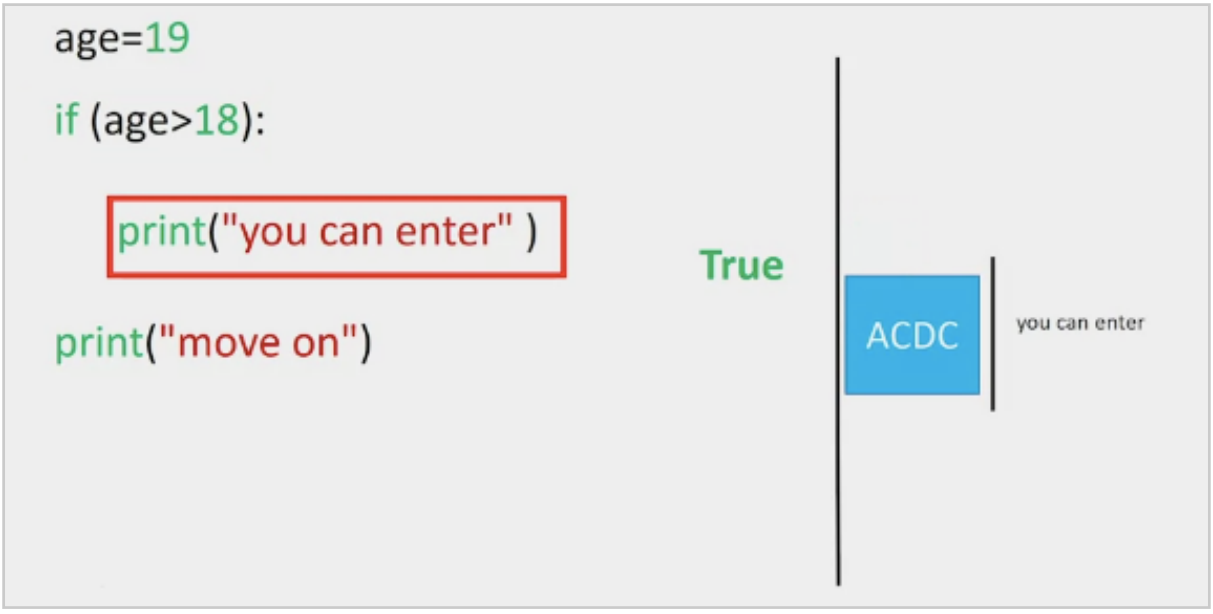
For example, consider the blue rectangle representing an ACDC concert. If the individual is 18 or older, they can enter the ACDC concert. If they are under the age of 18, they cannot enter the concert. Individual proceeds to the concert their age is 17, therefore, they are not granted access to the concert and they must move on. If the individual is 19, the condition is true. They can enter the concert then they can move on.

This is the syntax of the if statement from our previous example. We have the if statement.

We have the expression that can be true or false. The brackets are not necessary. We have a colon. Within an indent, we have the expression that is run if the condition is true. The statements after the if statement will run regardless if the condition is true or false. For the case where the age is 17, we set the value of the variable age to 17. We check the if statement, the statement is false. Therefore the program will not execute the statement to print, “you will enter”. In this case, it will just print “move on”.



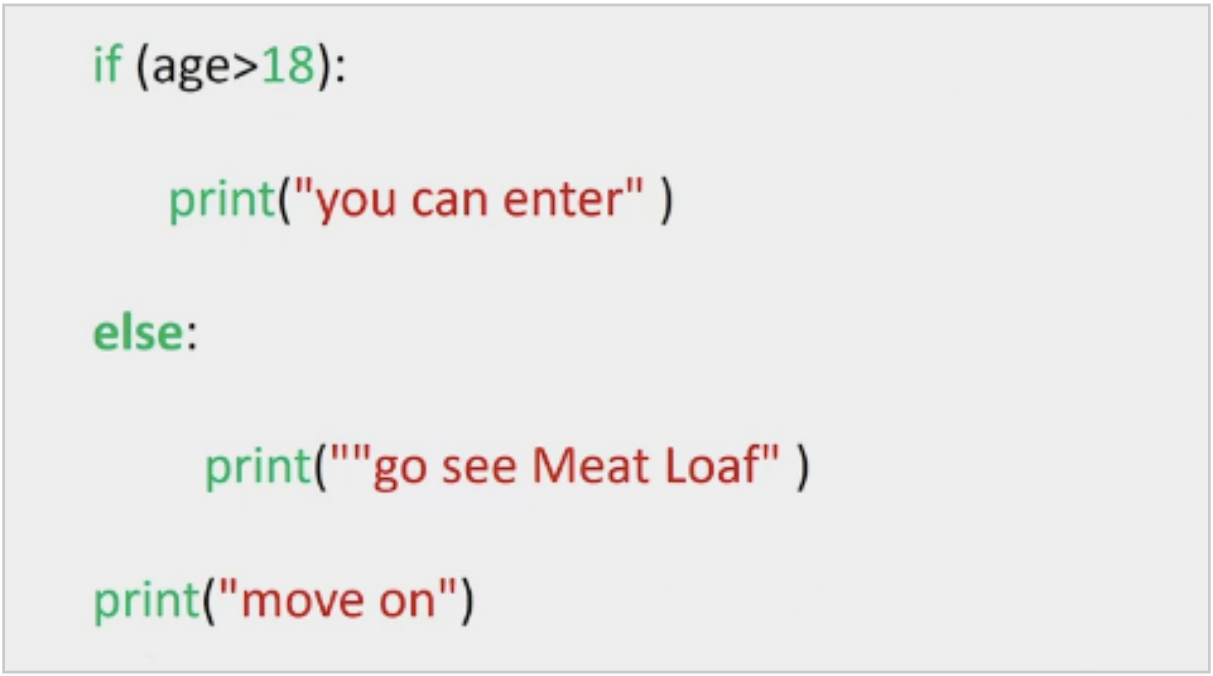
For the case where the age is 19, we set the value of the variable age to 19. We check the if statement. The statement is true. Therefore, the program will execute the statement to print “you will enter”. Then it will just print “move on”.



The `else` statement will run a different block of code if the same condition is false. Let’s use the ACDC concert analogy again. If the user is 17, they cannot go to the ACDC concert but they can go to the Meat Loaf concert represented by the purple square. If the individual is 19, the condition is true, they can enter the ACDC concert then they can move

on as before.

The syntax of the else statement is similar. We simply append the statement else.



We then add the expression we would like to execute with an indent. For the case where the age is 17, we set the value of the variable age to 17. We check the if statement, the statement is false. Therefore, we progress to the else statement. We run the statement in the indent. This corresponds to the individual attending the Meat Loaf concert. The program will then continue running.

For the case where the age is 19, we set the value of the variable age to 19. We check the if statement, the statement is true. Therefore, the program will execute the statement to print “you will enter”. The program skips the expressions in the else statement and continues to run the rest of the expressions.

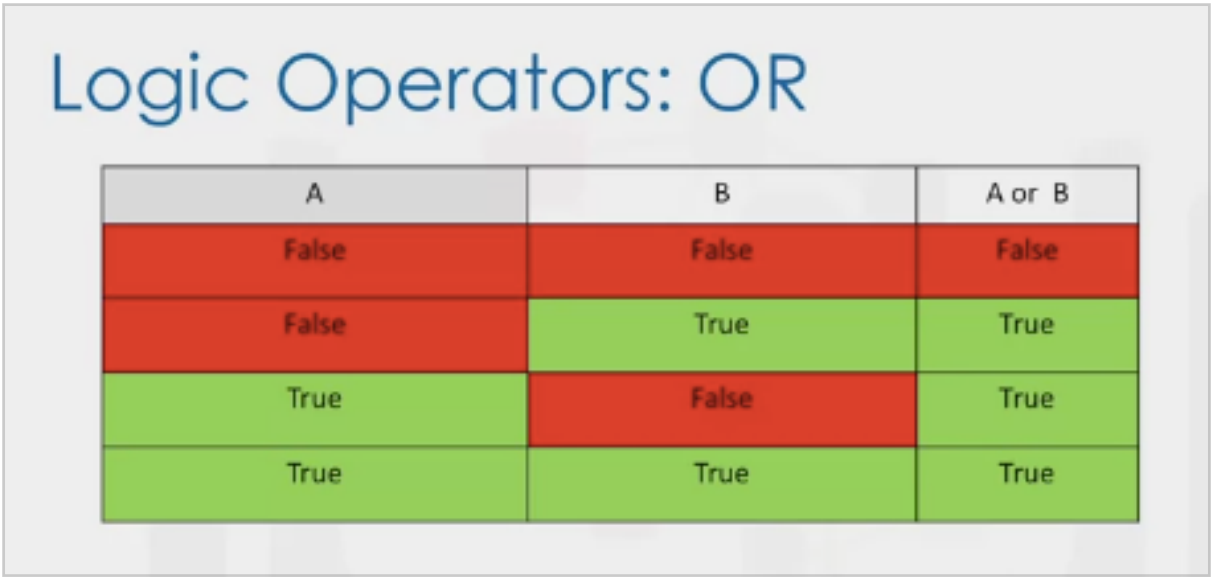
The `elif` statement, short for else if, allows us to check additional conditions if the preceding condition is false. If the condition is true, the alternate expressions will be run. Consider the concert example, if the individual is 18, they will go to the Pink Floyd concert instead of attending the ACDC or Meat Loaf concerts. The person of 18 years of age enters the area as they are not over 19 years of age. They cannot see ACDC but as they are 18 years, they attend Pink Floyd. After seeing Pink Floyd, they move on. The syntax of the elif statement is similar.

```
if (age>18):
    print("you can enter" )
elif(age==18):
        print("go see Pink Floyd" )
else:
        print(""go see Meat Loaf" )
print("move on")
```

We simply add the statement elif with the condition. We, then add the expression we would like to execute if the statement is true with an indent. Let's illustrate the code on the left. An 18 year old enters. They are not older than 18 years of age. Therefore, the condition is false. So the condition of the elif statement is checked. The condition is true. So then we would print "go see Pink Floyd". Then we would move on as before.
If the variable age was 17, the statement "go see Meat Loaf" would print. Similarly, if the age was greater than 18, the statement "you can enter" would print. Check the Labs for more examples.

Logic operators

Now let's take a look at logic operators. Logic operations take Boolean values and produce different Boolean values. The first operation is the `not` operator. If the input is true, the result is a false. Similarly, if the input is false, the result is a true.
Let A and B represent Boolean variables. The `OR` operator takes in the two values and produces a new Boolean value. We can use this table to represent the different values.

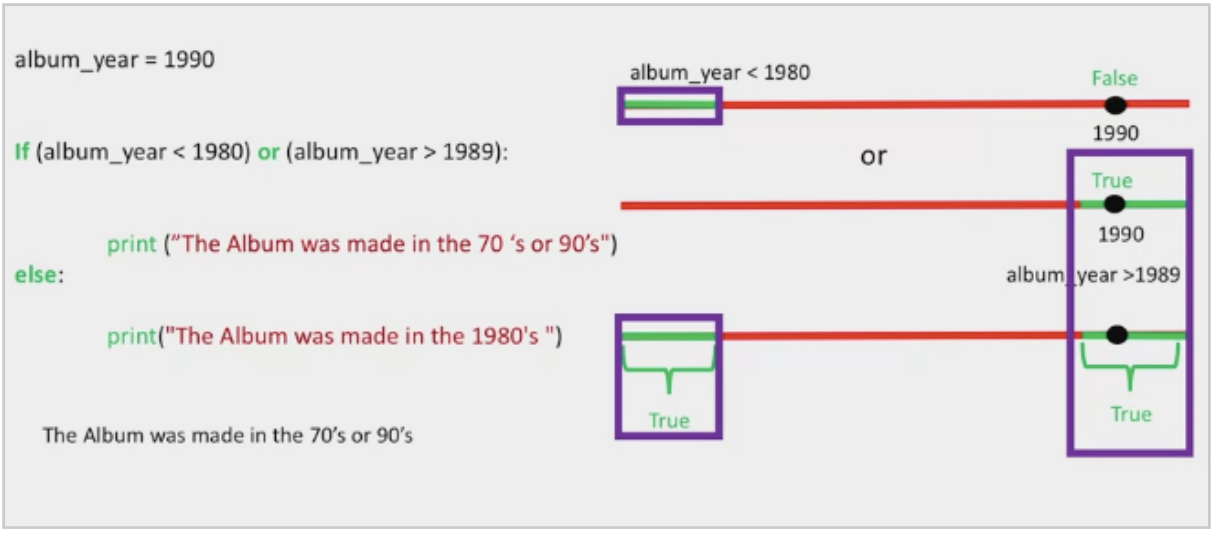


The first column represents the possible values of A. The second column represents the possible values of B. The final column represents the result of applying the OR operation. We see the **OR operator only produces a false if all the Boolean values are false**.

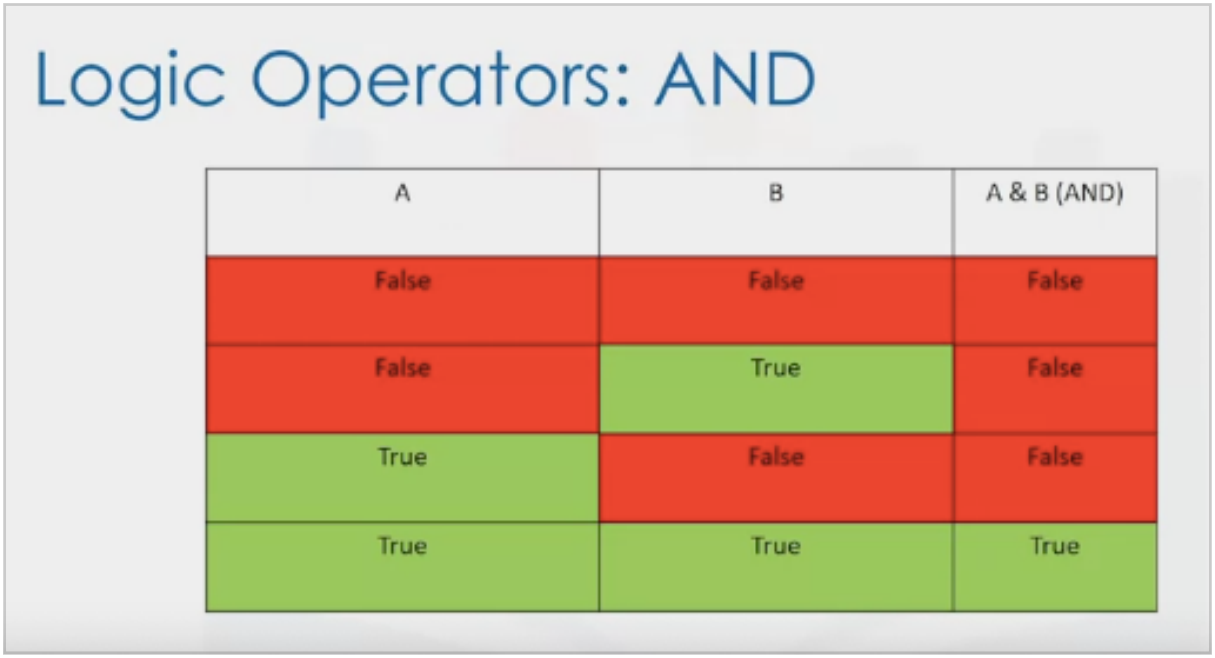
The following lines of code will print out: "This album was made in the 70s' or 90's", if the variable album year does not fall in the 80s.

```
album_year = 1990

If (album_year < 1980) or (album_year > 1989):

        print ("The Album was made in the 70 's or 90's")
else:

        print("The Album was made in the 1980's ")
```

Let's see what happens when we set the album year to 1990. The colored number line is green when the condition is true and red when the condition is false. In this case, the condition is false. Examining the second condition, we see that 1990 is greater than 1989. So the condition is true. We can verify by examining the corresponding second number line. In the final number line, the green region indicates, where the area is true. This region corresponds to where at least one statement is true. We see that 1990 falls in the area. Therefore, we execute the statement.



Let A and B represent Boolean variables. The `AND` operator takes in the two values and produces a new Boolean value. We can use this table to represent the different values.

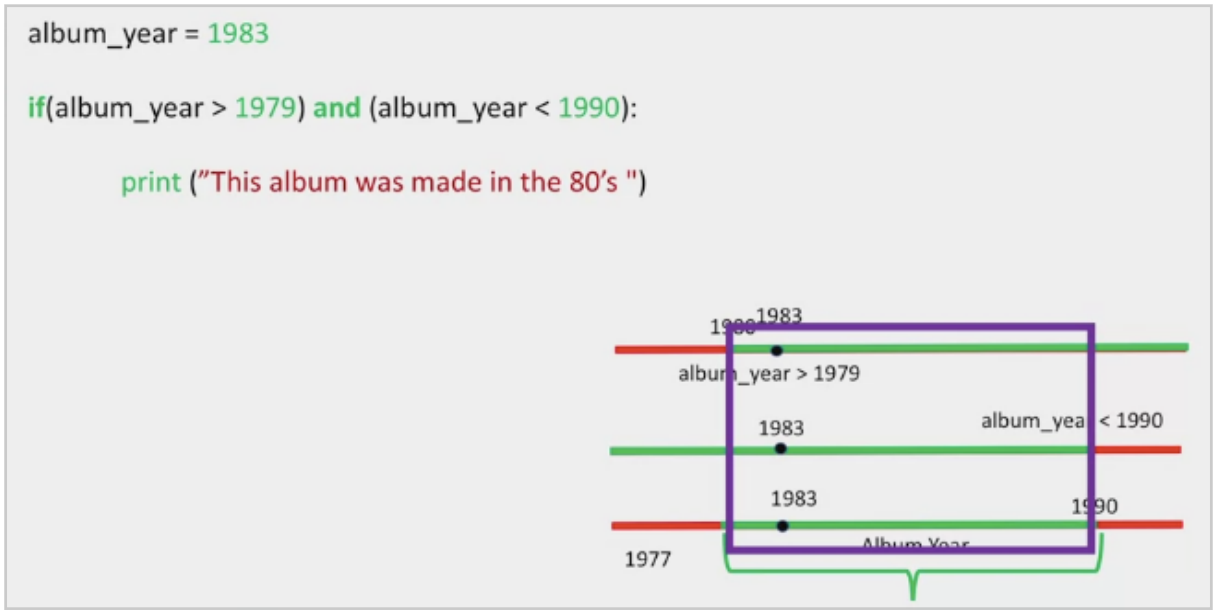


The first column represents the possible values of A. The second column represents the possible values of B. The final column represents the result of applying the AND operation. We see the **OR operator only produces a true if all the Boolean values are true**. The following lines of code will print out "This album was made in the 80's" if the variable album year is between 1980 and 1989.

```
album_year = 1983

if(album_year > 1979) and (album_year < 1990):

        print ("This album was made in the 80's ")
```

Let's see what happens when we set the album year to 1983. As before, we can use the

colored number line to examine where the condition is true. In this case, 1983 is larger than 1980, so the condition is true. Examining the second condition, we see that 1990 is greater than 1983. So, this condition is also true. We can verify by examining the corresponding second number line. In the final number line, the green region indicates where the area is true. Similarly, this region corresponds to where both statements are true. We see that 1983 falls in the area. Therefore, we execute the statement.



Branching allows us to run different statements for different inputs.

Lab - Conditions and Branching

Objectives

After completing this lab you will be able to:

- work with condition statements in Python, including operators, and branching.

Table of Contents

Condition Statements

Comparison Operators

Comparison operations compare some value or operand and based on a condition, produce a Boolean. When comparing two values you can use these operators:

- equal: `==`
- not equal: `!=`
- greater than: `>`
- less than: `<`
- greater than or equal to: `>=`
- less than or equal to: `<=`

Let's assign `a` a value of 5. Use the equality operator denoted with two equal `==` signs to determine if two values are equal. The case below compares the variable `a` with 6.

```
# Condition Equal
a = 5
a == 6
```

```
False
```

The result is False, as 5 does not equal to 6.

Consider the following equality comparison operator: `i > 5`. If the value of the left operand, in this case the variable `i` is greater than the value of the right operand, in this case 5, then the statement is True. Otherwise, the statement is False. If `i` is equal to 6, because 6 is larger than 5, the output is True.

```
# Greater than Sign
i = 6
i > 5
```

```
True
```

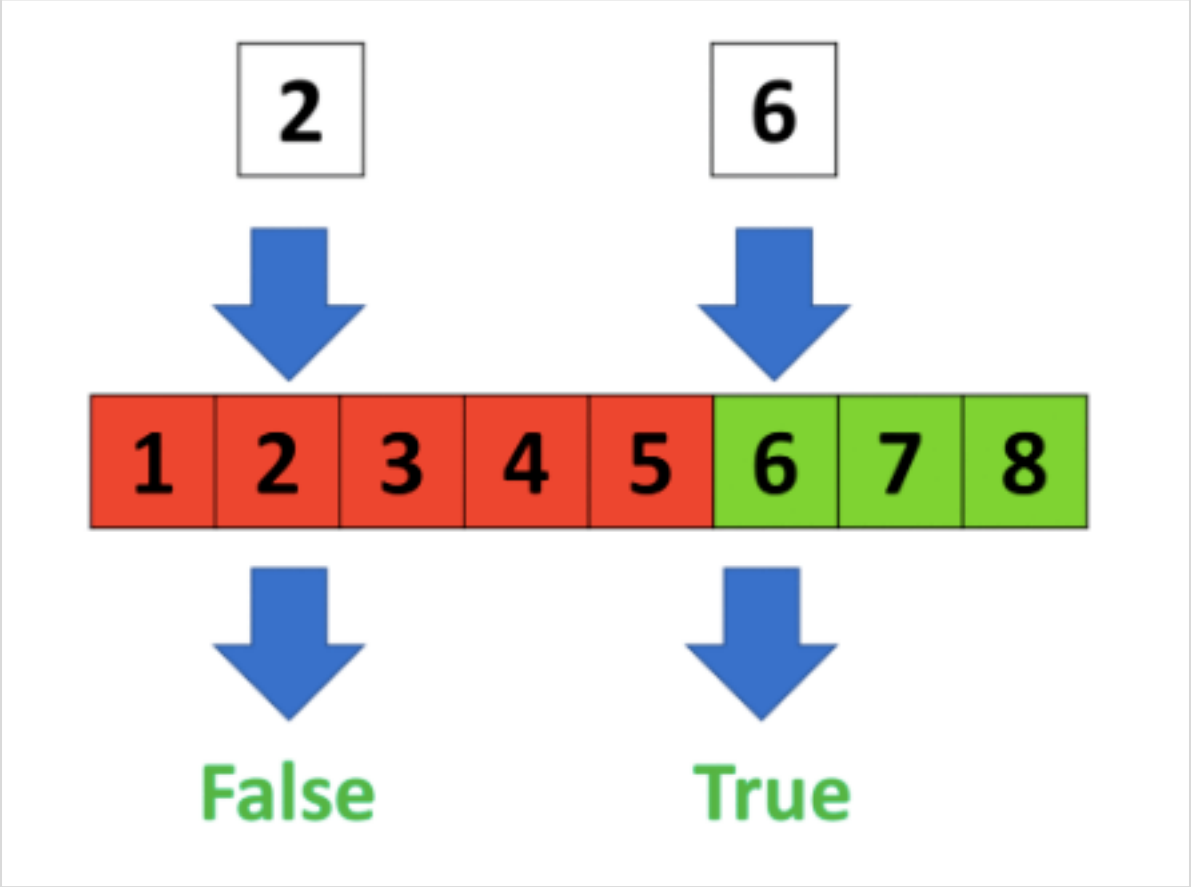
Set `i = 2`. The statement is False as 2 is not greater than 5:

```
# Greater than Sign
i = 2
i > 5
```

```
```

False

Let's display some values for `i` in the figure. Set the values greater than 5 in green and the rest in red. The green region represents where the condition is **True**, the red where the statement is **False**. If the value of `i` is 2, we get **False** as the 2 falls in the red region. Similarly, if the value for `i` is 6 we get a **True** as the condition falls in the green region.



The inequality test uses an exclamation mark preceding the equal sign, if two operands are not equal then the condition becomes **True**. For example, the following condition will produce **True** as long as the value of `i` is not equal to 6:

```
# Inequality Sign
i = 2
i != 6
```

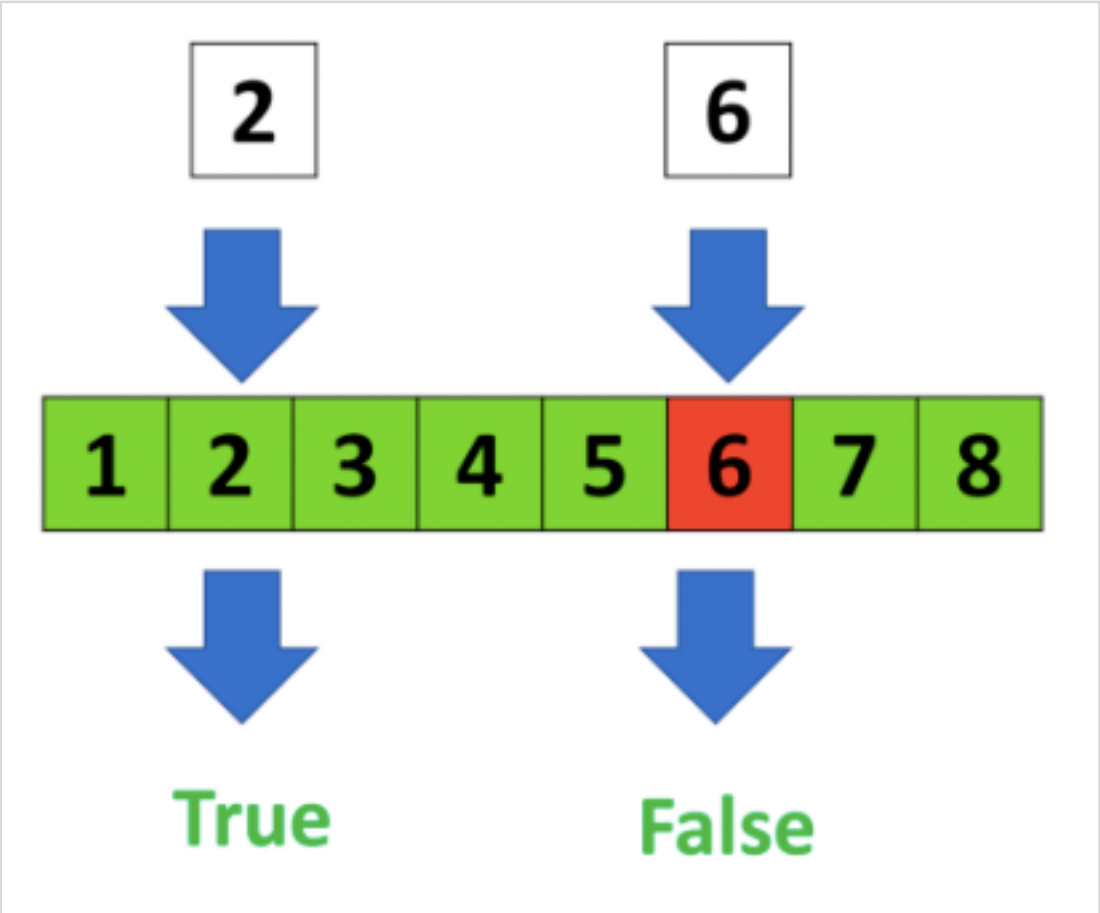
True

When `i` equals 6 the inequality expression produces False.

```
# Inequality Sign
i = 6
i != 6
```

```
False
```

See the number line below. When the condition is **True**, the corresponding numbers are marked in green and for where the condition is **False** the corresponding number is marked in red. If we set `i` equal to 2 the operator is true, since 2 is in the green region. If we set `i` equal to 6, we get a **False**, since the condition falls in the red region.



We can apply the same methods on strings. For example, we can use an equality operator on two different strings. As the strings are not equal, we get a **False**.

```
```

```
# Use Equality sign to compare the strings
"ACDC" == "Michael Jackson"
```

False

If we use the inequality operator, the output is going to be **True** as the strings are not equal.

```
# Use Inequality sign to compare the strings
"ACDC" != "Michael Jackson"
```

True

The inequality operation is also used to compare the letters/words/symbols according to the ASCII value of letters. The decimal value shown in the following table represents the order of the character:

Char.	ASCII	Char.	ASCII	Char.	ASCII	Char.	ASCII
A	65	N	78	a	97	n	110
B	66	O	79	b	98	o	111
C	67	P	80	c	99	p	112
D	68	Q	81	d	100	q	113
E	69	R	82	e	101	r	114
F	70	S	83	f	102	s	115
G	71	T	84	g	103	t	116
H	72	U	85	h	104	u	117
I	73	V	86	i	105	v	118
J	74	W	87	j	106	w	119
K	75	X	88	k	107	x	120
L	76	Y	89	l	108	y	121
M	77	Z	90	m	109	z	122

For example, the ASCII code for ! is 33, while the ASCII code for + is 43. Therefore + is larger than ! as 43 is greater than 33.

Similarly, from the table above we see that the value for A is 65, and the value for B is 66, therefore:

```
# Compare characters
'B' > 'A'
```

True

When there are multiple letters, the first letter takes precedence in ordering:

```
# Compare characters
```

```
'BA' > 'AB'
```

True

Note: Upper Case Letters have different ASCII code than Lower Case Letters, which means the comparison between the letters in Python is case-sensitive.

Branching

Branching allows us to run different statements for different inputs. It is helpful to think of an **if statement** as a locked room, if the statement is **True** we can enter the room and your program will run some predefined tasks, but if the statement is **False** the program will ignore the task.

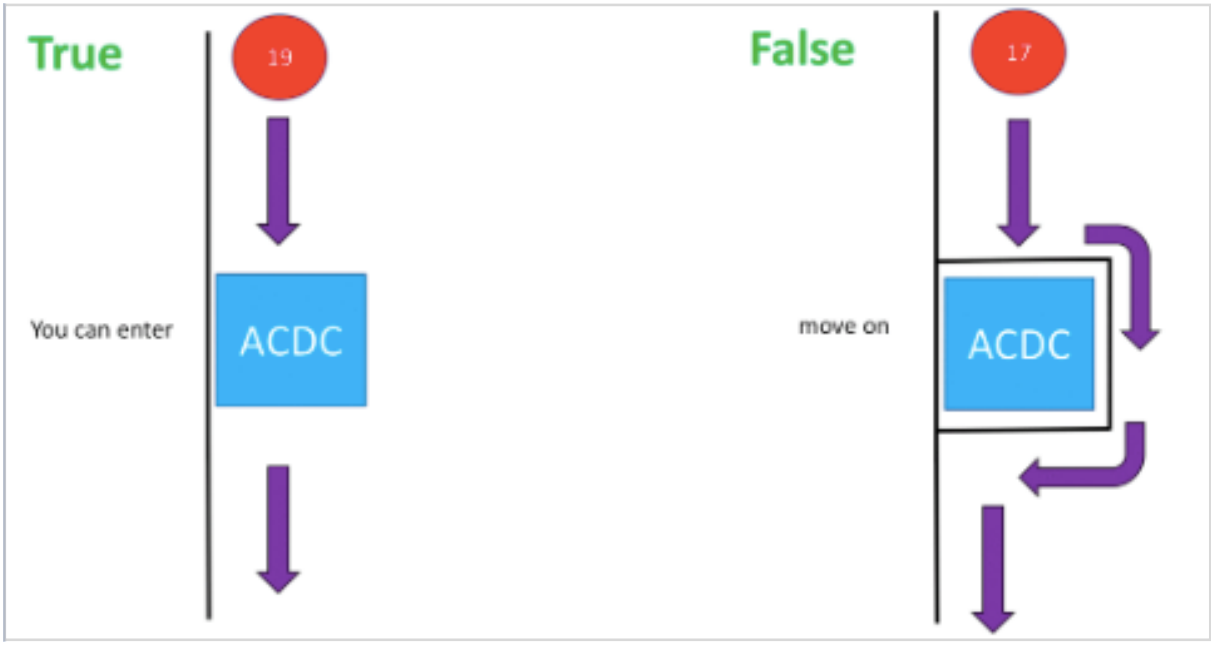
For example, consider the blue rectangle representing an ACDC concert. If the individual is older than 18, they can enter the ACDC concert. If they are 18 or younger, they cannot enter the concert.

We can use the condition statements learned before as the conditions that need to be checked in the **if statement**. The syntax is as simple as `if *condition statement :`, which contains a word `if`, any condition statement, and a colon at the end. Start your tasks which need to be executed under this condition in a new line with an indent. The lines of code after the colon and with an indent will only be executed when the **if statement** is **True**. The tasks will end when the line of code does not contain the indent. In the case below, the code `print("you can enter")` is executed only if the variable `age` is greater than 18 is a True case because this line of code has the indent. However, the execution of `print("move on")` will not be influenced by the if statement.

```
# If statement example
age = 19
#age = 18

#expression that can be true or false
if age > 18:
    #within an indent, we have the expression that is run if the condition is
true
    print("you can enter" )
#The statements after the if statement will run regardless if the condition is
```

```
true or false
print("move on")



    you can enter
    move on
```

It is helpful to use the following diagram to illustrate the process. On the left side, we see what happens when the condition is True. The person enters the ACDC concert representing the code in the indent being executed; they then move on. On the right side, we see what happens when the condition is False; the person is not granted access, and the person moves on. In this case, the segment of code in the indent does not run, but the rest of the statements are run.

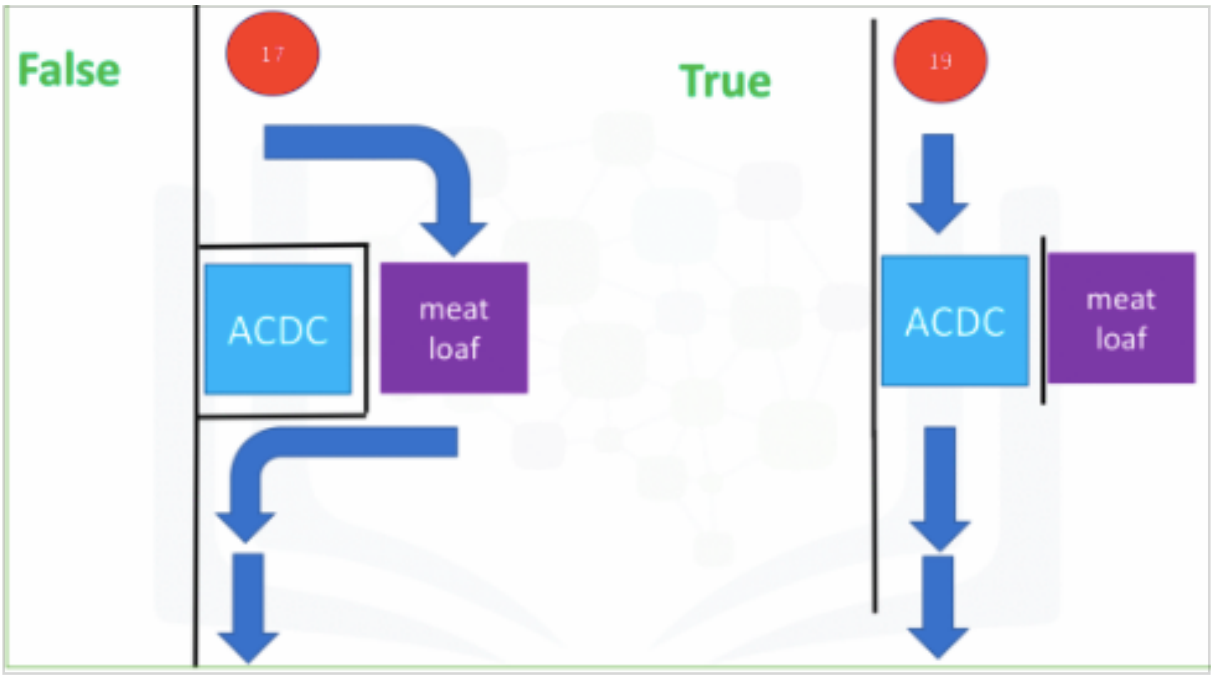


The `else` statement runs a block of code if none of the conditions are **True** before this `else` statement. Let's use the ACDC concert analogy again. If the user is 17 they cannot go to the ACDC concert, but they can go to the Meatloaf concert. The syntax of the `else` statement is similar as the syntax of the `if` statement, as `else :`. Notice that, there is no condition statement for `else`. Try changing the values of `age` to see what happens:

```
# Else statement example


age = 18
```

```
# age = 19
if age > 18:
    print("you can enter" )
else:
    print("go see Meat Loaf" )
print("move on")



    go see Meat Loaf
    move on
```

The process is demonstrated below, where each of the possibilities is illustrated on each side of the image. On the left is the case where the age is 17, we set the variable age to 17, and this corresponds to the individual attending the Meatloaf concert. The right portion shows what happens when the individual is over 18, in this case 19, and the individual is granted access to the concert.

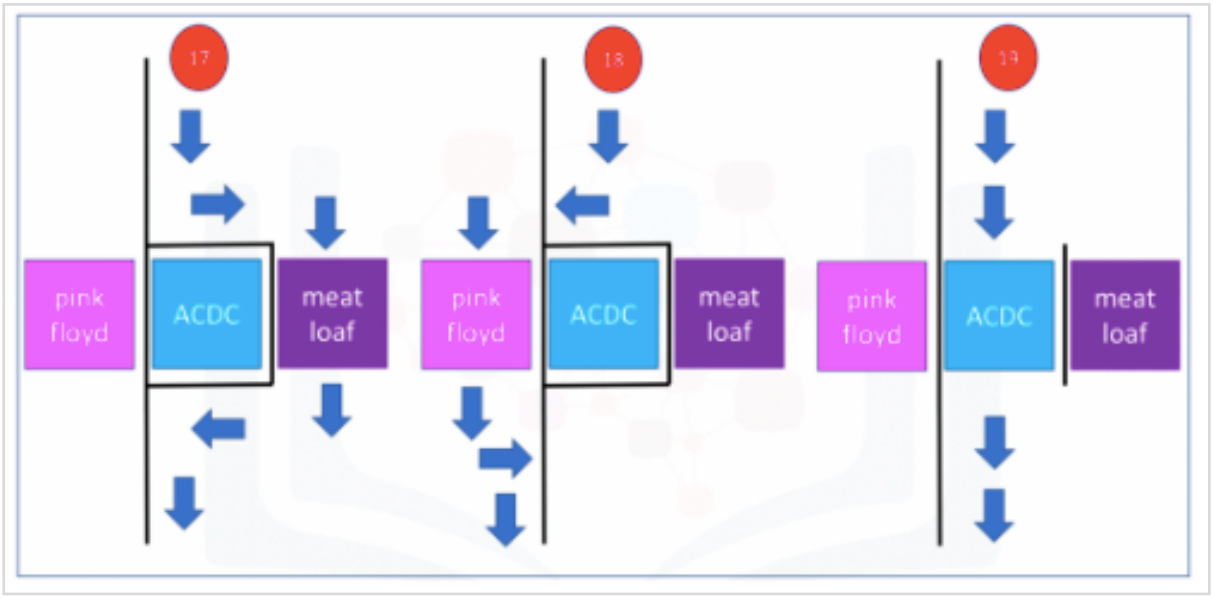


The `elif` statement, short for else if, allows us to check additional conditions if the condition statements before it are False. If the condition for the `elif` statement is True, the alternate expressions will be run. Consider the concert example, where if the individual is 18 they will go to the Pink Floyd concert instead of attending the ACDC or Meat-loaf concert. A person that is 18 years of age enters the area, and as they are not older than 18

they can not see ACDC, but since they are 18 years of age, they attend Pink Floyd. After seeing Pink Floyd, they move on. The syntax of the `elif` statement is similar in that we merely change the `if` in the `if` statement to `elif`.

```
# Elif statment example

age = 18

if age > 18:
    print("you can enter" )
elif age == 18:
    print("go see Pink Floyd")
else:
    print("go see Meat Loaf" )
print("move on")



    go see Pink Floyd
    move on
```

The three combinations are shown in the figure below. The left-most region shows what happens when the individual is less than 18 years of age. The central component shows when the individual is exactly 18. The rightmost shows when the individual is over 18.



Look at the following code:

```
# Condition statement example
album_year = 1983
#album_year = 1970
if album_year > 1980:
    print("Album year is greater than 1980")
print('do something..')
```

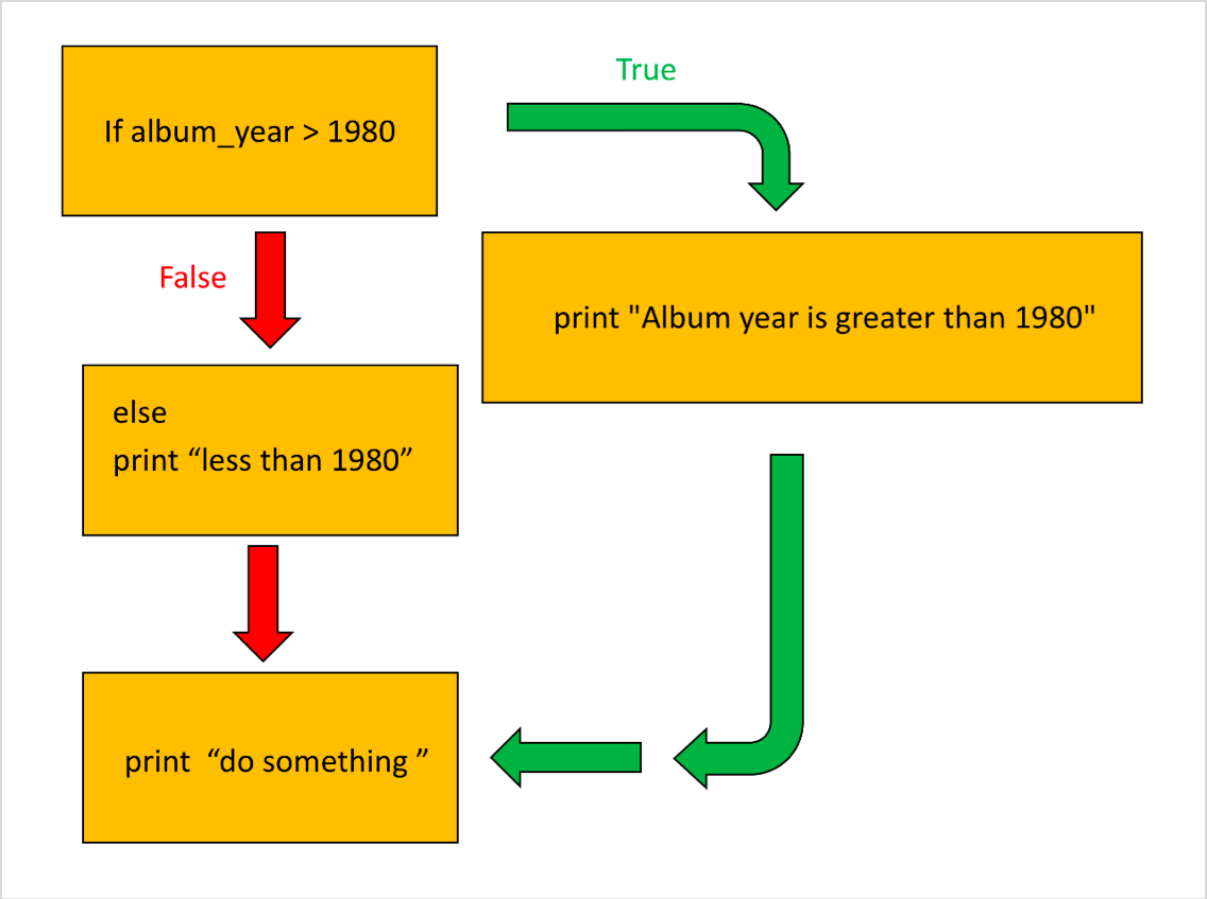
```
    Album year is greater than 1980
    do something..
```

Feel free to change `album_year` value to other values -- you'll see that the result changes! Notice that the code in the above indented block will only be executed if the results are True.
As before, we can add an `else` block to the `if` block. The code in the `else` block will only be executed if the result is False.

Syntax:

```
if (condition):
  do something
else:
  do something else
```

If the condition in the `if` statement is False, the statement after the `else` block will execute. This is demonstrated in the figure:




```
# Condition statement example

album_year = 1983
#album_year = 1970

if album_year > 1980:
    print("Album year is greater than 1980")
else:
    print("less than 1980")

print('do something..')
```

```
    Album year is greater than 1980
    do something..
```

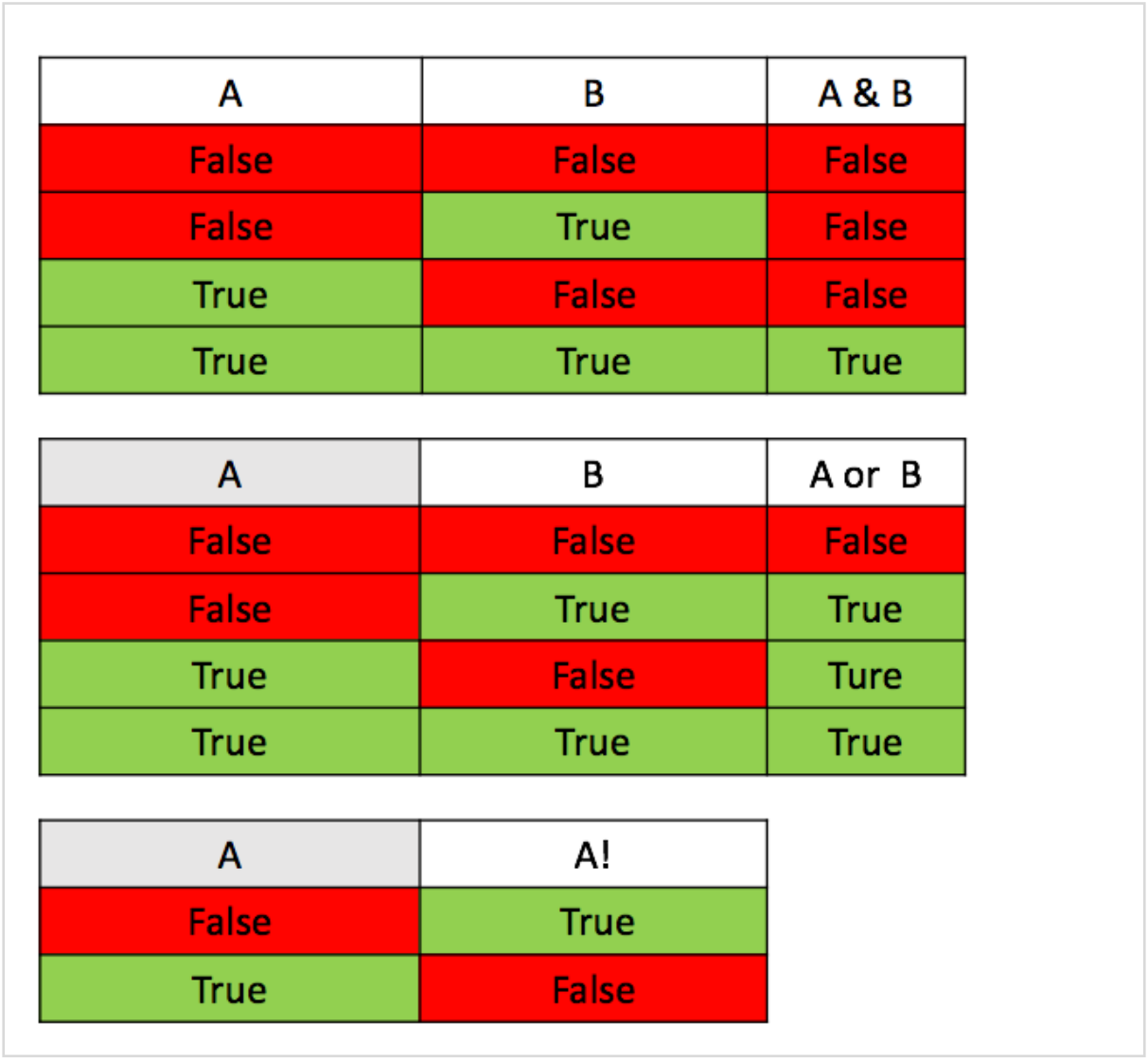
Feel free to change the `album_year` value to other values -- you'll see that the result changes based on it!

Logical operators

Sometimes you want to check more than one condition at once. For example, you might want to check if one condition and another condition are both **True**. Logical operators allow you to combine or modify conditions.

- `and`
- `or`
- `not`

These operators are summarized for two variables using the following truth tables:



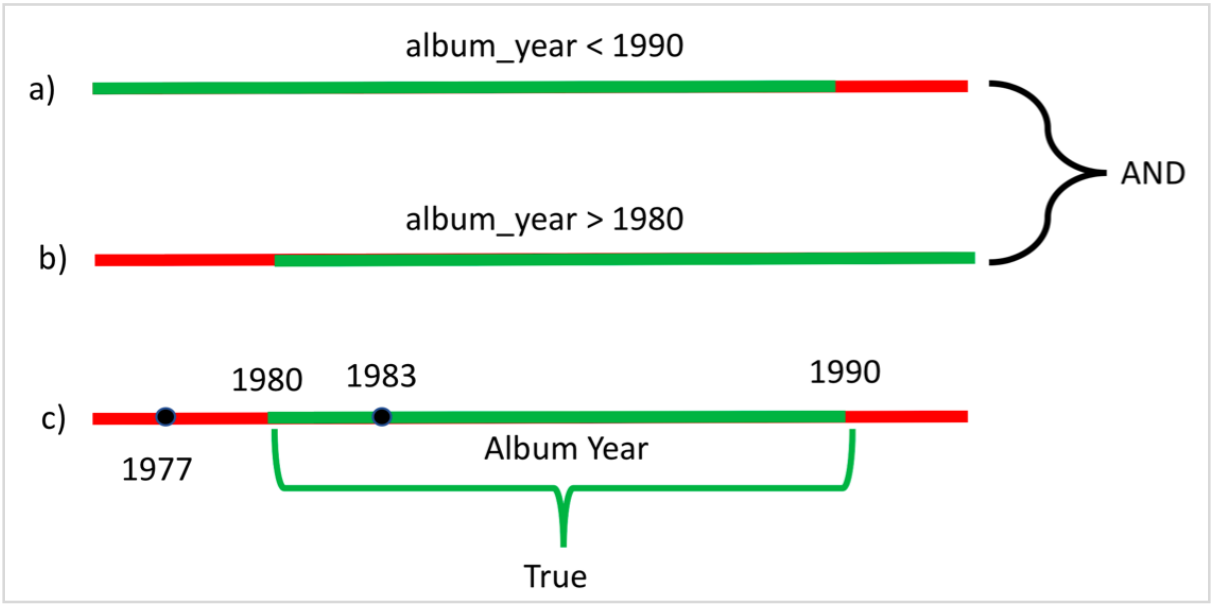
A	B	A & B
False	False	False
False	True	False
True	False	False
True	True	True

A	B	A or B
False	False	False
False	True	True
True	False	Ture
True	True	True

A	A!
False	True
True	False

The `and` statement is only **True** when both conditions are true. The `or` statement is True if one condition, or both are **True**. The `not` statement outputs the opposite truth value.

Let's see how to determine if an album was released after 1979 (1979 is not included) and before 1990 (1990 is not included). The time periods between 1980 and 1989 satisfy this condition. This is demonstrated in the figure below. The green on lines **a** and **b** represents periods where the statement is **True**. The green on line **c** represents where both conditions are **True**, this corresponds to where the green regions overlap.



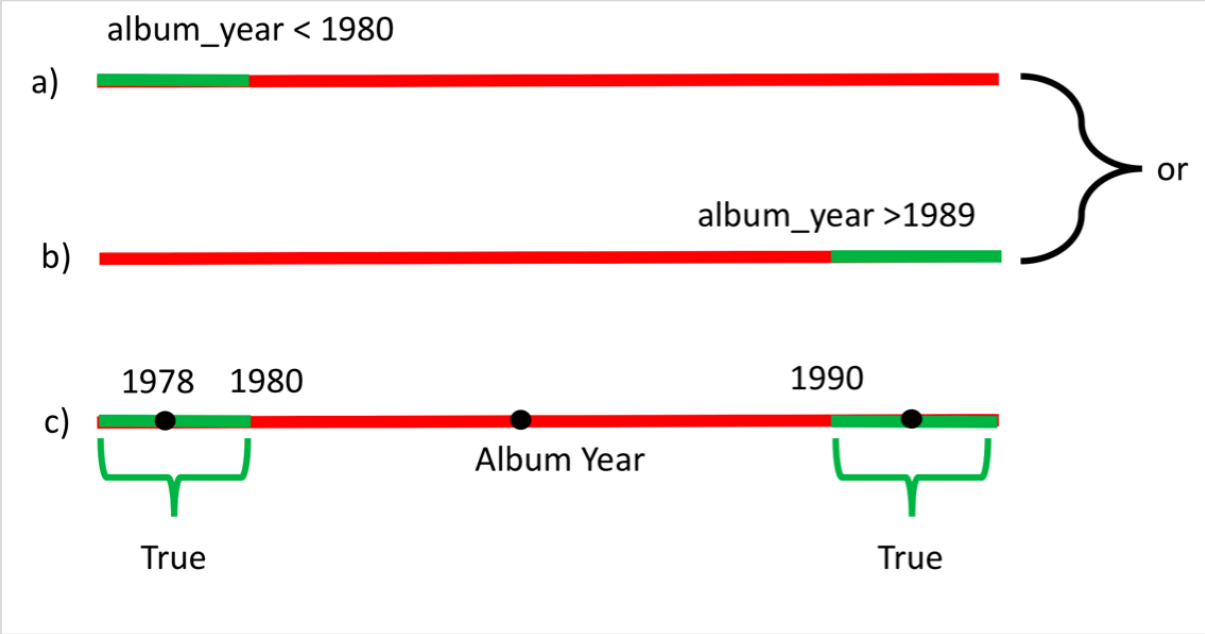
The block of code to perform this check is given by:

```
# Condition statement example
album_year = 1980
if(album_year > 1979) and (album_year < 1990):
    print ("Album year was in between 1980 and 1989")
print("")
print("Do Stuff..")
```

```
    Album year was in between 1980 and 1989

    Do Stuff..
```

To determine if an album was released before 1980 (1979 and earlier) or after 1989 (1990 and onward), an or statement can be used. Periods before 1980 (1979 and earlier) or after

1989 (1990 and onward) satisfy this condition. This is demonstrated in the following figure, the color green in **a** and **b** represents periods where the statement is true. The color green in **c** represents where at least one of the conditions
are true.



The block of code to perform this check is given by:

```
# Condition statement example
album_year = 1990

if(album_year < 1980) or (album_year > 1989):
    print ("Album was not made in the 1980's")
else:
    print("The Album was made in the 1980's ")
```

Album was not made in the 1980's

The `not` statement checks if the statement is false:

```
# Condition statement example
album_year = 1983
if not (album_year == '1984'):
```

```
    print ("Album year is not 1984")
```

Album year is not 1984

Quiz on Conditions

Write an if statement to determine if an album had a rating greater than 8. Test it using the rating for the album “Back in Black” that had a rating of 8.5. If the statement is true print "This album is Amazing!"

```
# Write your code below and press Shift+Enter to execute
album_rating = 8.5

if album_rating > 8:
    print("This album is Amazing")
else:
    print("This album sucks! ><")
```

This album is Amazing

Write an if-else statement that performs the following. If the rating is larger then eight print “this album is amazing”. If the rating is less than or equal to 8 print “this album is ok”.

```
# Write your code below and press Shift+Enter to execute
album_rating = 8

if album_rating > 8:
    print("This album is Amazing")
else:
    print("This album sucks! ><")
```

This album sucks! ><

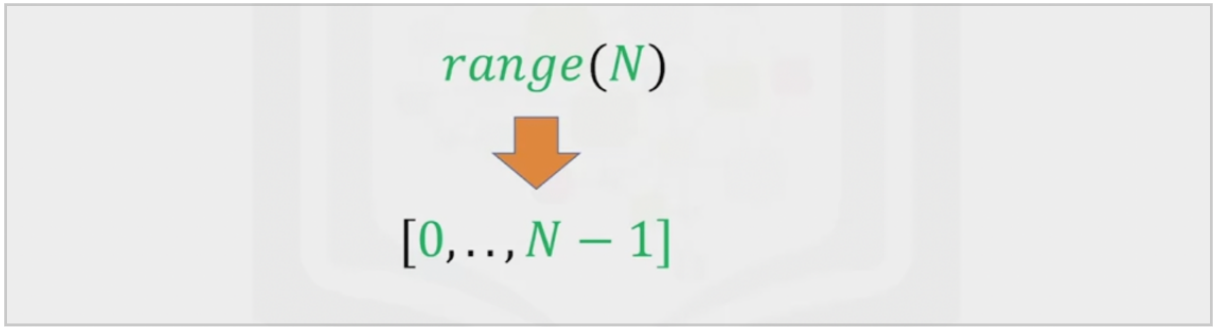
Write an if statement to determine if an album came out before 1980 or in the years: 1991 or 1993. If the condition is true print out the year the album came out.

```
# Write your code below and press Shift+Enter to execute
years = 1979
if years < 1980 or years == 1991 or years == 1993:
    print("This album came out in year",years)
```

This album came out in year 1979

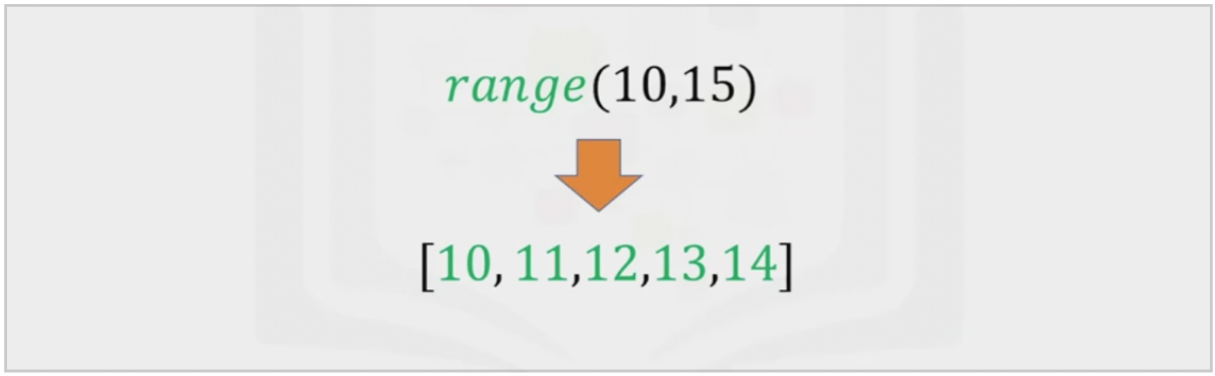
Loops theory

In this video we will cover Loops in particular `for` loops and `while` loops. We will use many visual examples in this video. See the labs for examples with data. Before we talk about loops, let's go over the `range` function.
The range function outputs and ordered sequence as a list I. If the input is a positive integer, the output is a sequence. The sequence contains the same number of elements as the input but starts at zero.



For example, if the input is three the output is the sequence zero, one, two.

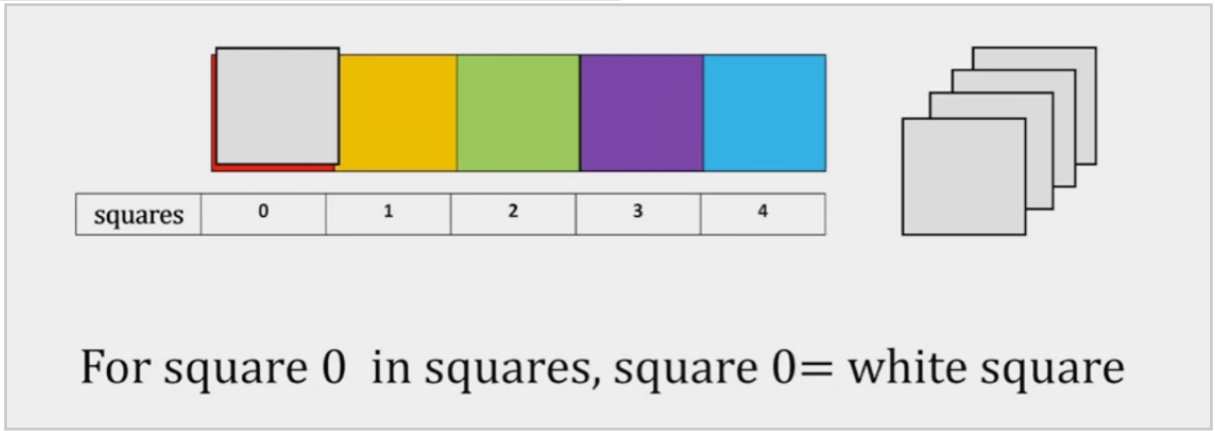
If the range function has two inputs where the first input is smaller than the second input, the output is a sequence that starts at the first input. Then the sequence iterates up to but not including the second number. For the input 10 and 15 we get the following sequence. See the labs for more capabilities of the range function.



Please note, if you use Python three, the range function will not generate a list explicitly like in Python two.

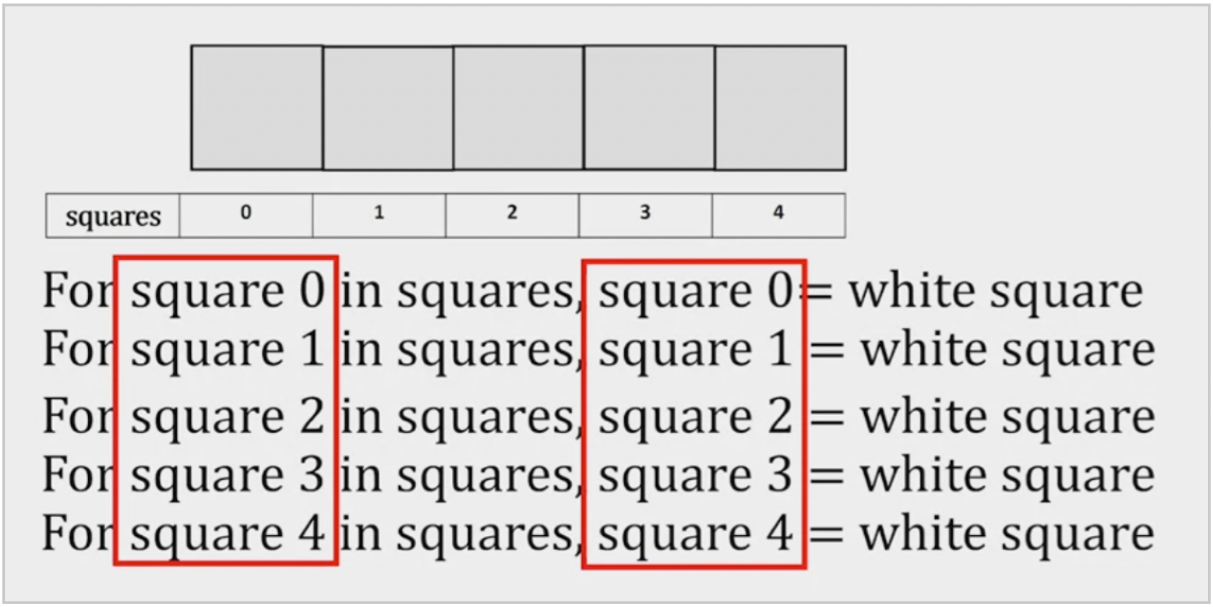
In this section, we will cover `for loops`. We **will focus on lists,** but many of the procedures can be used on tuples. Loops perform a task over and over. Consider the group of colored squares.

Let's say we would like to replace each colored square with a white square. Let's give each square a number to make things a little easier and refer to all the group of squares as squares. If we wanted to tell someone to replace squares zero with a white square, we would say equals replace square zero with a white square or we can say `for squares zero in squares square zero equals white square`.

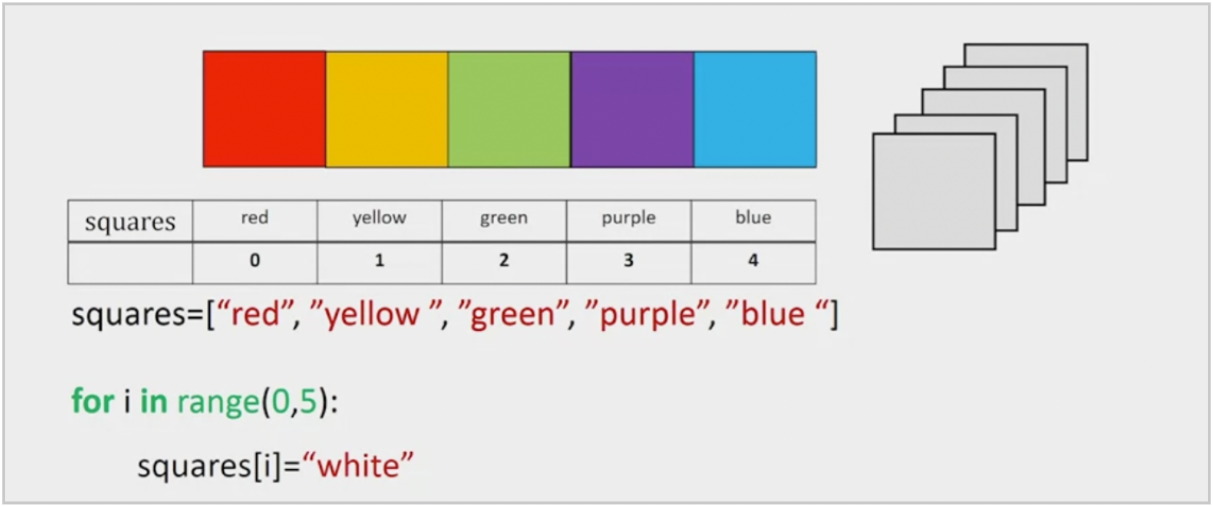


Similarly, for the next square we can say for square one in squares, square one equals white square. For the next square we can say for square two in squares, square two equals

white square. We repeat the process for each square. The only thing that changes is the index of the square we are referring to.



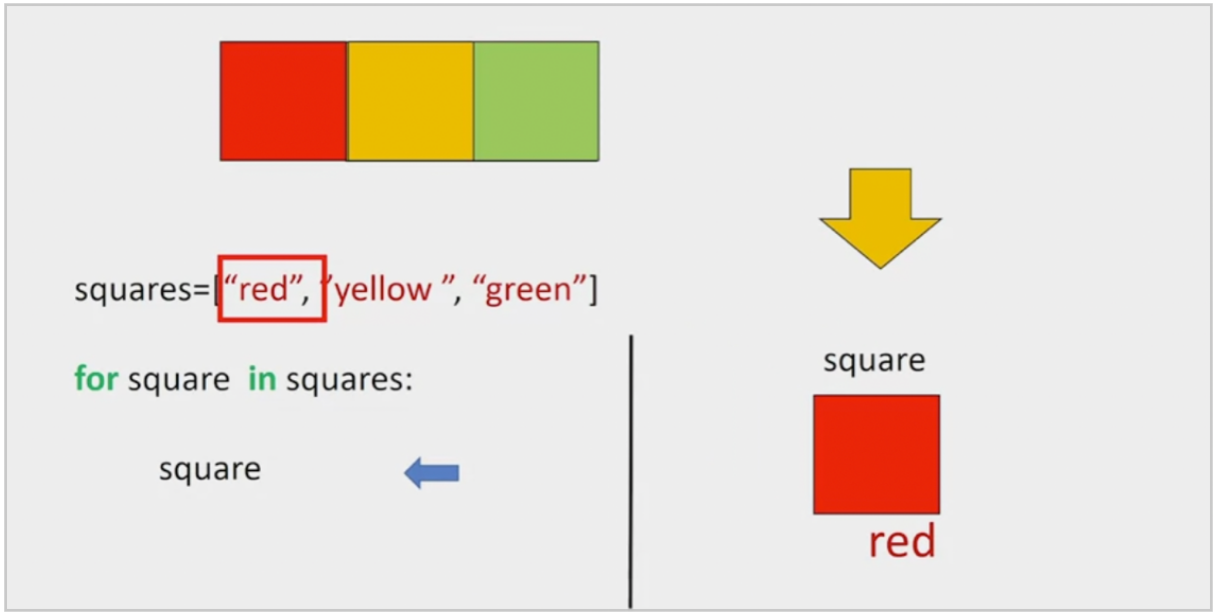
If we're going to perform a similar task I**n Python we cannot use actual squares**. So let's use a list to represent the boxes. Each element in the list is a string representing the color. We want to change the name of the color in each element to white. Each element in the list has the following index. This is a syntax to perform a loop in Python.



Notice the indent, the range function generates a list. The code will simply repeat everything in the indent five times. If you were to change the value to six it would do it 6 times. However, the value of I is incremented by one each time. In this segment we change the I element of the list to the string white. The value of I is set to zero. **Each iteration of the loop starts at the beginning of the indent**. We then run everything in the indent. The first element in the list is set to white. We then go to the start of the indent, we progress down each line.

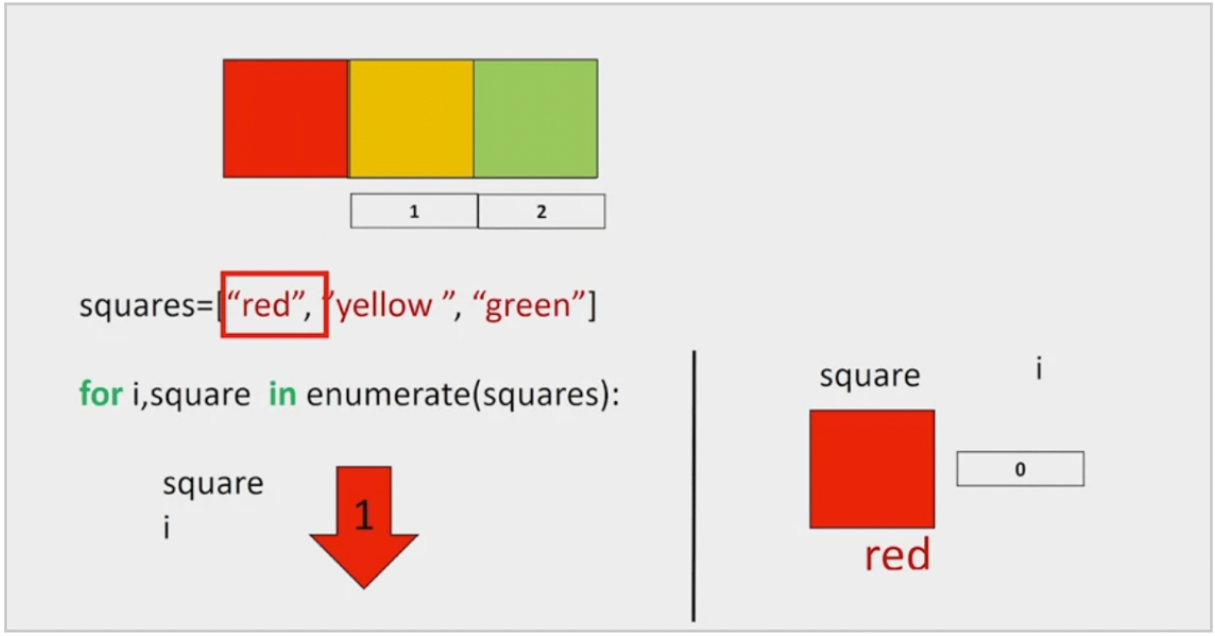
When we reach the line to change the value of the list, we set the value of index `1` to white. The value of I increases by one. We repeat the process for index `2`. The process continues for the next index, until we've reached the final element.

We can also iterate through a **list** or **tuple** directly in python, **we do not even need to use indices**. Here is the list squares. Each iteration of the list we pass one element of the list squares to the variable square. Lets display the value of the variable square on this section.



For the first iteration, the value of square is red, we then start the second iteration. For the second iteration, the value of square is yellow. We then start the third iteration. For the final iteration, the value of square is green.

A useful function for iterating data is `enumerate`. It can be used to obtain the index and the element in the list. Let's use the box analogy with the numbers representing the index of each square. This is the syntax to iterate through a list and provide the index of each element.



We use the list squares and use the names of the colors to represent the colored squares. The argument of the function enumerate is the list. In this case squares the variable `i` is the **index** and the variable `square` is the corresponding element in the list.

Let's use the left part of the screen to display the different values of the variable square

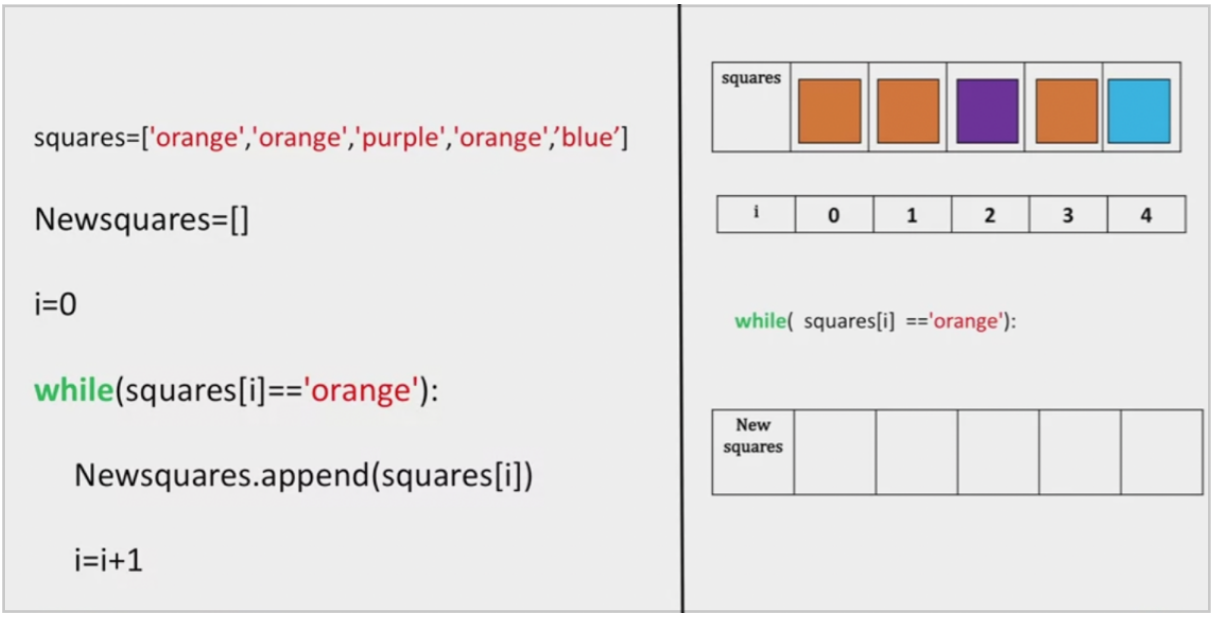
and I for the various iterations of the loop:

- For the first iteration, the value of the variable is red corresponding to the zeroth index, and the value for I is zero.
- For the second iteration the value of the variable square is yellow, and the value of I corresponds to its index i.e. 1.
- We repeat the process for the last index.

`While loops` are similar to for loops but instead of executing a statement a set number of times a **while loop will only run if a condition is met**.

Let's say we would like to copy all the orange squares from the list squares to the list New squares. But we would like to stop if we encounter a non-orange square. We don't know the value of the squares beforehand. We would simply continue the process while the square is orange or see if the square equals orange. If not, we would stop. For the first example, we would check if the square was orange. It satisfies the conditions so we would copy the square. We repeat the process for the second square. The condition is met. So we copy the square. In the next iteration, we encounter a purple square. The condition is not met. So we stop the process. This is essentially what a while loop does.

Let's use the figure on the left to represent the code.



We will use a list with the names of the color to represent the different squares. We create an empty list of new squares. In reality the list is of indeterminate size. We start the index at zero the while statement will repeatedly execute the statements within the indent until the condition inside the bracket is false. We append the value of the first element of the list squares to the list new squares. We increase the value of I by one. We append the value of the second element of the list squares to the list new squares. We increment the value of I. Now the value in the array squares is purple; therefore, the condition for the while statement is false and we exit the loop.

Loops LAB

Objectives

After completing this lab you will be able to:

- work with the loop statements in Python, including for-loop and while-loop.
-

Loops in Python

This notebook will teach you about the loops in the Python Programming Language. By the end of this lab, you'll know how to use the loop statements in Python, including for loop, and while loop.

Table of Contents

Loops

Range

Sometimes, you might want to repeat a given operation many times. Repeated executions like this are performed by **loops**. We will look at two types of loops, `for` loops and `while` loops.

Before we discuss loops lets discuss the `range` object. It is helpful to think of the range object as an ordered list. For now, let's look at the simplest case. If we would like to generate an object that contains elements ordered from 0 to 2 we simply use the following command:

```
# Use the range
range(3)
```

```
range(0, 3)
```

NOTE: While in Python 2.x it returned a list as seen in video lessons, in 3.x it returns a range object.

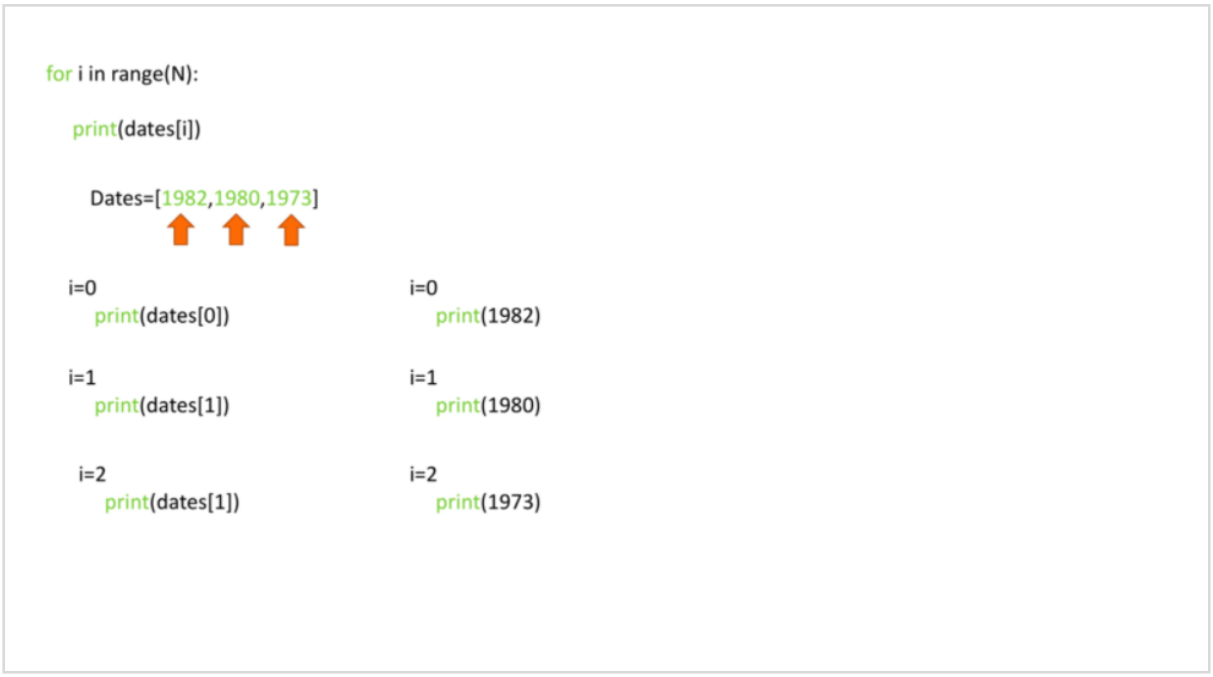
What is `for` loop?
The `for` loop enables you to execute a code block multiple times. For example, you would use this if you would like to print out every element in a list.\
Let's try to use a `for` loop to print all the years presented in the list `dates`:

This can be done as follows:

```
# For loop example
dates = [1982,1980,1973]
N = len(dates)

for i in range(N):
    print(dates[i])
```

1982
1980
1973

The code in the indent is executed `N` times, each time the value of `i` is increased by 1 for every execution. The statement executed is to `print` out the value in the list at index `i` as shown here:



In this example we can print out a sequence of numbers from 0 to 7:

```
# Example of for loop
for i in range(0, 8):
    print(i)
```

```
0
1
2
3
4
5
6
7
```

In Python we can **directly access** the elements in the list as follows:

```
# Exmaple of for loop, loop through list

for year in dates:
    print(year)
```

```
1973
1973
1973
```

For each iteration, the value of the variable `year` behaves like the value of `dates[i]` in the first example:

We can **change** the elements in a list:

```
# Use for loop to change the elements in list

squares = ['red', 'yellow', 'green', 'purple', 'blue']

for i in range(0, 5):
    print("Before square ", i, 'is',  squares[i])
    squares[i] = 'white'
    print("After square ", i, 'is',  squares[i])
```

```
Before square  0 is red
After square  0 is white
Before square  1 is yellow
After square  1 is white
Before square  2 is green
After square  2 is white
Before square  3 is purple
After square  3 is white
Before square  4 is blue
After square  4 is white
```

We can **access** the **index** and the **elements** of a list as follows:

```
# Loop through the list and iterate on both index and element value
squares=['red', 'yellow', 'green', 'purple', 'blue']

for i, square in enumerate(squares):
    print(i, square)
```

```
0 red
1 yellow
2 green
3 purple
4 blue
```

What is `while` loop?

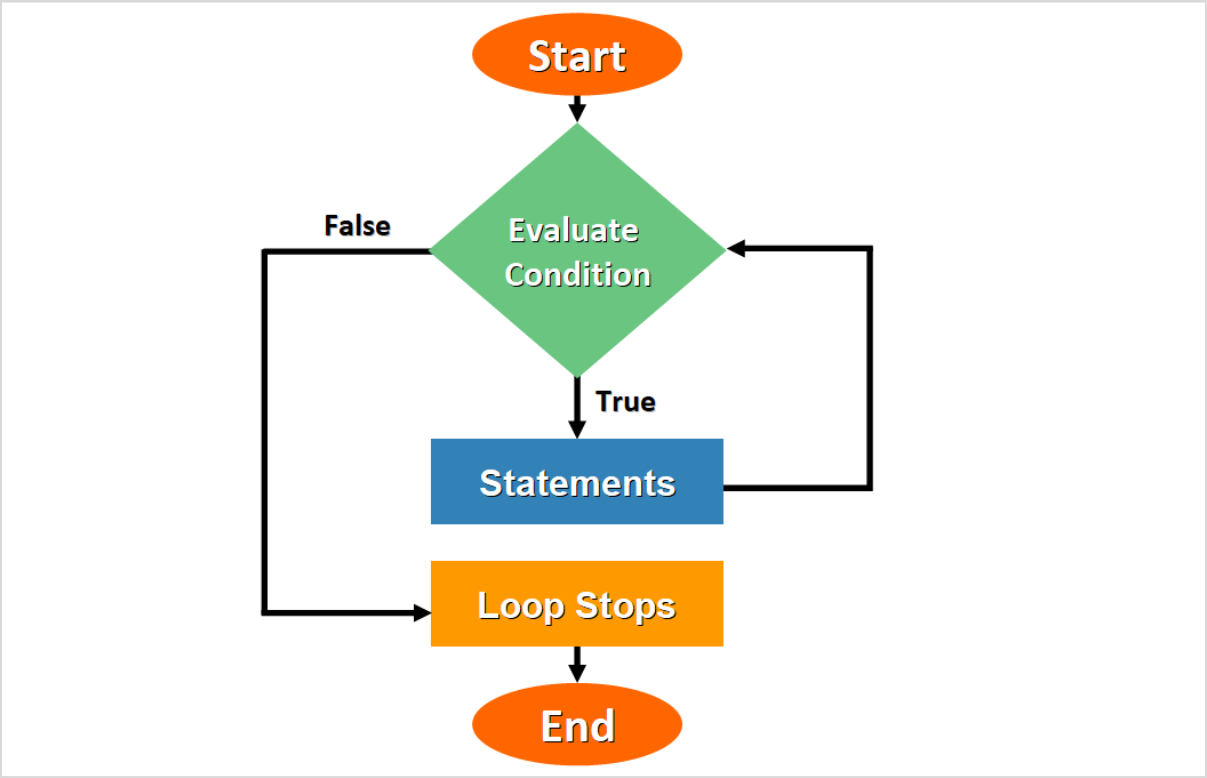
As you can see, the `for` loop is used for a controlled flow of repetition. However, what if we don't know when we want to stop the loop? What if we want to keep executing a code block until a certain condition is met? The `while` loop exists as a tool for **repeated execution based on a condition**. The code block will keep being executed until the given logical condition returns a **False** boolean value.

Let's say we would like to iterate through list `dates` and stop at the year 1973, then print out the number of iterations. This can be done with the following block of code:

```
# While Loop Example
dates = [1982, 1980, 1973, 2000]

i = 0
year = dates[0]

while(year != 1973):
    print(year)
    i = i + 1
    year = dates[i]

print("It took ", i ,"repetitions to get out of loop.")
```

```
#Da notare che il metodo print è fuori dal ciclo while



    1982
    1980
    It took  2 repetitions to get out of loop.
```

A while loop iterates merely until the condition in the argument is not met, as shown in the following figure:



Quiz on Loops

Write a `for` loop the prints out all the element between **-5** and **5** using the range function.

```
# Write your code below and press Shift+Enter to execute
for i in range(-4,5):
    print(i)
```

```
-4
-3
-2
-1
0
1
2
3
4
```

Print the elements of the following list:
`Genres = [ 'rock', 'R\&B', 'Soundtrack', 'R\&B', 'soul', 'pop']`
Make sure you follow Python conventions.

```
# Write your code below and press Shift+Enter to execute
Genres=[ 'rock', 'R&B', 'Soundtrack', 'R&B', 'soul', 'pop']
for i in Genres:
    print(i)
```

```
rock
R&B
Soundtrack
R&B
soul
pop
```

Write a for loop that prints out the following list:
`squares=['red', 'yellow', 'green', 'purple', 'blue']`

```
# Write your code below and press Shift+Enter to execute
squares=['red', 'yellow', 'green', 'purple', 'blue']
```

```
for i in squares:
    print(i)
```

```
red
yellow
green
purple
blue
```

Write a while loop to display the values of the Rating of an album playlist stored in the list `PlayListRatings`. If the score is less than 6, exit the loop. The list `PlayListRatings` is given by:

`PlayListRatings = [10, 9.5, 10, 8, 7.5, 5, 10, 10]`

```
# Write your code below and press Shift+Enter to execute

PlayListRatings = [10, 9.5, 10, 8, 7.5, 5, 10, 10]
i = 0
rating = PlayListRatings[i]

while(i < len(PlayListRatings) and rating >= 6):
    print(rating)
    i = i + 1
    rating= PlayListRatings[i]
# NB. Se questa riga viene inserita prima viene stampato anche il 5, non
capisco perché, come si può vedere di seguito.
```

```
10
9.5
10
8
7.5
```

```
PlayListRatings = [10, 9.5, 10, 8, 7.5, 5, 10, 10]

i = 0
Rating = PlayListRatings[0]

while(Rating >= 6):
    Rating = PlayListRatings[i]
    print(Rating)
    i = i + 1



    10
    9.5
    10
    8
    7.5
    5
```

Write a while loop to copy the strings `'orange'` of the list `squares` to the list `new_squares`. Stop and exit the loop if the value on the list is not `'orange'`:

```
# Write your code below and press Shift+Enter to execute

squares = ['orange', 'orange', 'purple', 'blue ', 'orange']
new_squares = []

i = 0
color = squares[0]

while(color == 'orange'):
    new_squares.append(color)
    i = i + 1
```

```
    color = squares[i]
print(new_squares)
```

['orange', 'orange']

NB il metodo Print deve essere inserito al di fuori del ciclo per stampare correttamente la nuova lista.

```
# Stesso esempio con un codice più semplice

squares = ['orange', 'orange', 'purple', 'blue ', 'orange']
new_squares = []

i = 0

while(squares[i] == 'orange'):
    new_squares.append(squares[i])
    i = i +1
print(new_squares)
```

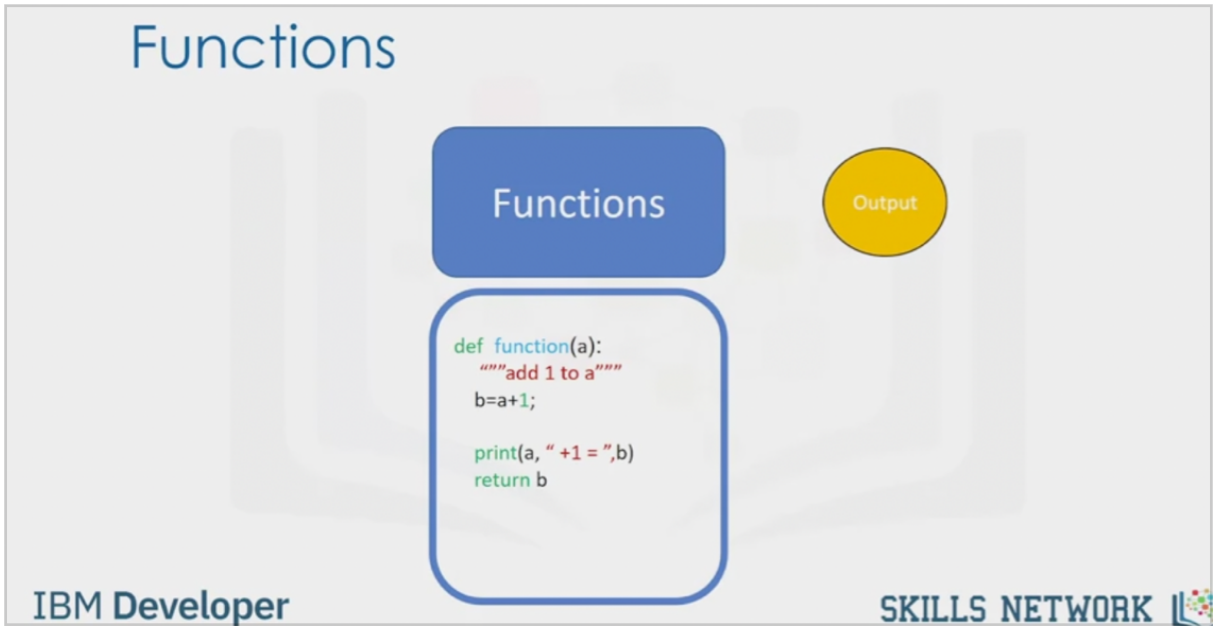
['orange', 'orange']

```
squares = ['orange', 'orange', 'purple', 'blue ', 'orange']
new_squares = []
i = 0
while(i < len(squares) and squares[i] == 'orange'):
    new_squares.append(squares[i])
    i = i + 1
print (new_squares)
```

['orange', 'orange']

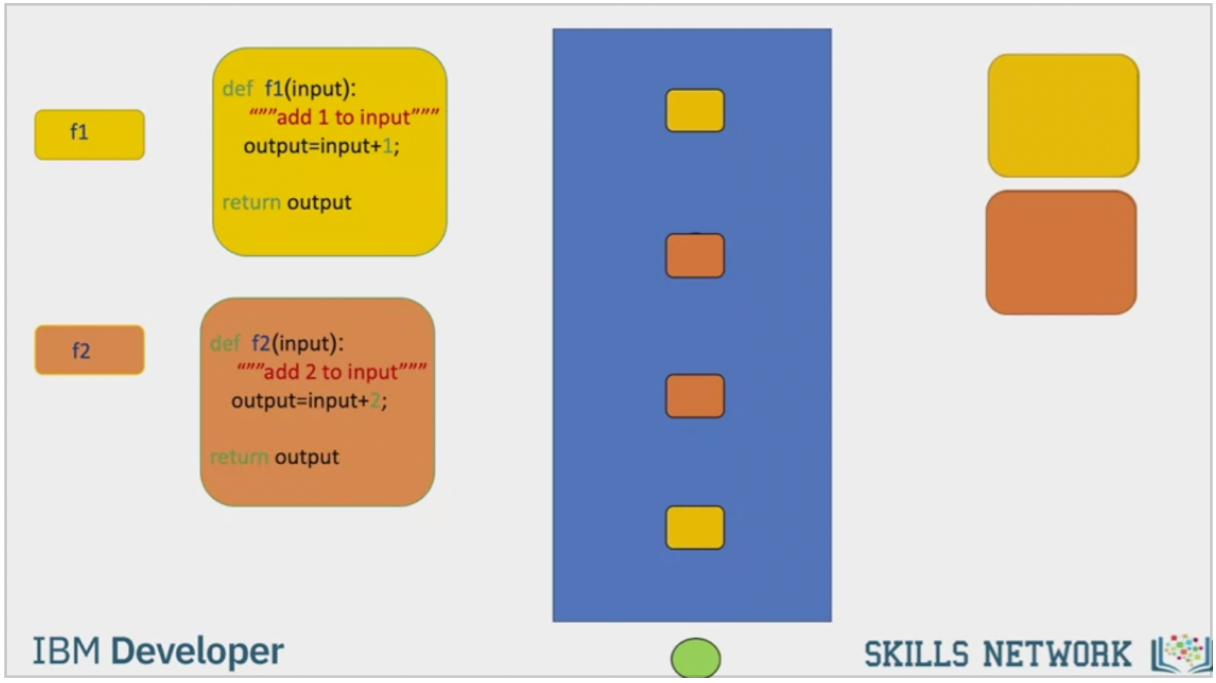
Functions theory

In this video we will cover functions. You will learn how to use some of Python's built-in functions as well as how to build your own functions. **Functions take some input then produce some output or change**. The function is just a piece of code you can reuse.



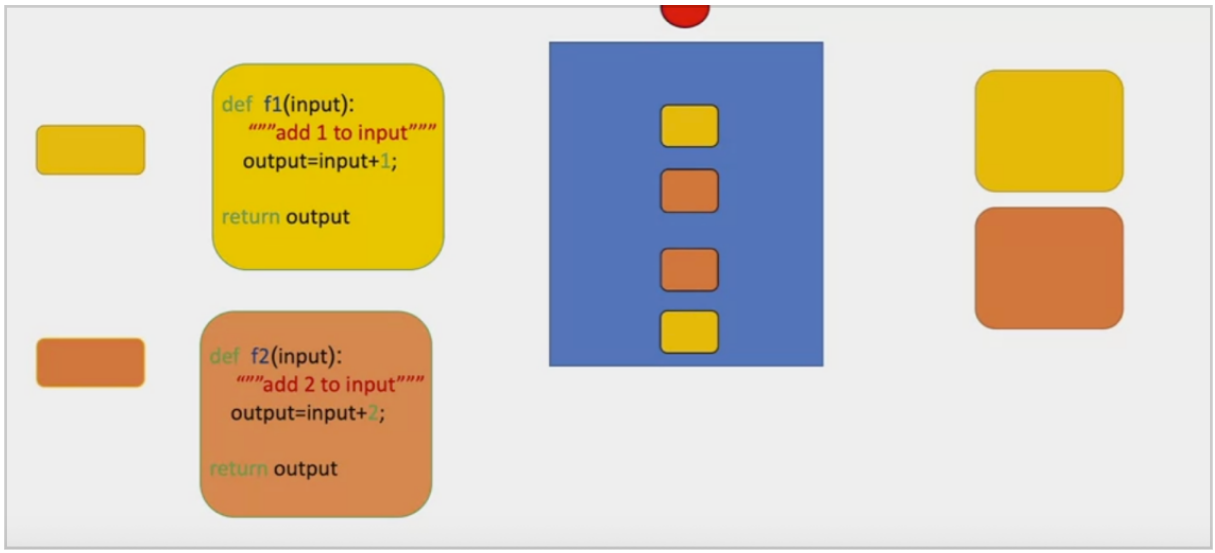
You can implement your own function, but in many cases, you use other people's functions. In this case, **you just have to know how the function works and in some cases how to import the functions**.

Let the orange and yellow squares represent similar blocks of code.



We can run the code using some input and get an output. **If we define a function to do the task we just have to call the function**. Let the small squares represent the lines of code used to call the function. We can replace these long lines of code by just calling the

function a few times. Now we can just call the function; our code is much shorter.



The code performs the same task.

You can think of the process like this:

- when we call the function f1, we pass an input to the function. These values are passed to all those lines of code you wrote.
- This returns a value; you can use the value. For example, you can input this value to a new function f2. When we call this new function f2, the value is passed to another set of lines of code. The function returns a value.
- The process is repeated passing the values to the function you call. You can save these functions and reuse them, or use other people's functions.

Python has many built-in functions; you don't have to know how those functions work internally, but simply what task those functions perform. The function `len` takes in an input of type sequence, such as a string or list, or type collection, such as a dictionary or set, and returns the length of that sequence or collection. Consider the following list. The len function takes this list as an argument, and we assign the result to the variable L. The function determines there are 8 items in the list, then returns the length of the list, in this case, 8.

The function `sum` takes in an iterable like a **tuple** or **list** and returns the total of all the elements. Consider the following list. We pass the list into the sum function and assign the result to the variable S. The function determines the total of all the elements, then returns it, in this case, the value is 70.

There are two ways to **sort** a list. The first is using the function `sorted`. We can also use the list method sort. Methods are similar to functions. Let's use this as an example to illustrate the difference.

- The function `sorted` Returns a new sorted list or tuple. Consider the list album ratings. We can apply the function sorted to the list album ratings and get a new list sorted album rating. The result is a new sorted list. If we look at the list album ratings, nothing has changed. Generally, functions take an input, in this case, a list. They produce a new

output, in this instance, a sorted list.

- If we use the method `sort`, **the list** album ratings **will change and no new list will be created**. Let's use this diagram to help illustrate the process. In this case, the rectangle represents the list album ratings. When we apply the method sort to the list, the list album rating changes. Unlike the previous case, we see that the list album rating has changed. In this case, no new list is created.

Now that we have gone over how to use functions in Python, let's see **how to build** our own functions. We will now get you started on building your own functions in python. This is an example of a function in python that returns its input value + 1.

- To define a function, we start with the keyword def. The name of the function should be descriptive of what it does.
- We have the function formal parameter "A" in parentheses, followed by a colon.
- We have a code block with an **indent**, for this case, we add 1 to "A" and assign it to B.
- We return or output the value for b.

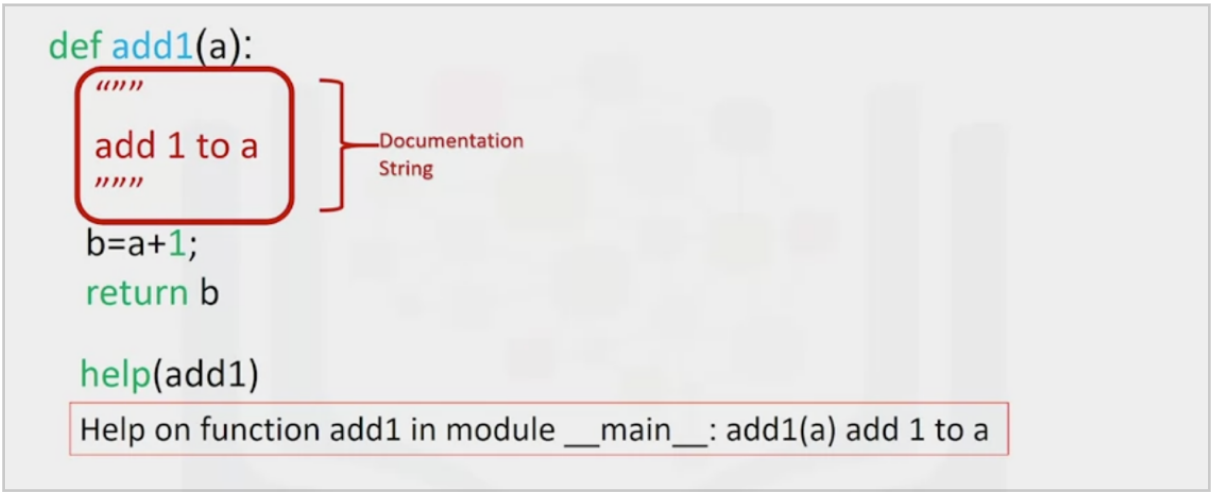
```
def add1(a):
    b=a+1
    return b

add1(5)
6
c=add1(10)
c:11
```

After we define the function, we can call it. The function will add 1 to 5 and return a 6. We can call the function again; this time assign it to the variable "c" The value for 'c' is 11. Let's explore this further. Let's go over an example when you call a function. It should be noted that this is a simplified model of Python, and Python does not work like this under the hood.

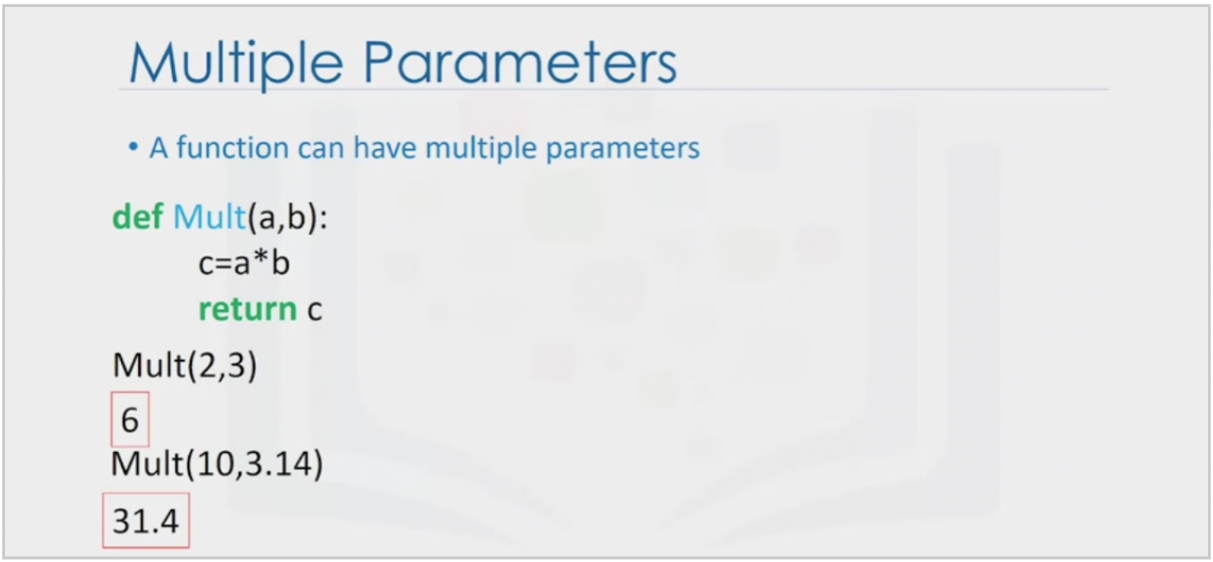
1. We call the function giving it an input, 5. It helps to think of the value of 5 as being passed to the function.
2. Now the sequences of commands are run, the value of "A" is 5.
3. "B" would be assigned a value of 6.
4. We then return the value of b, in this case, as b was assigned a value of 6, the function returns a 6.

If we call the function again, the process starts from scratch; we pass in an 8. The subsequent operations are performed. Everything that happened in the last call will

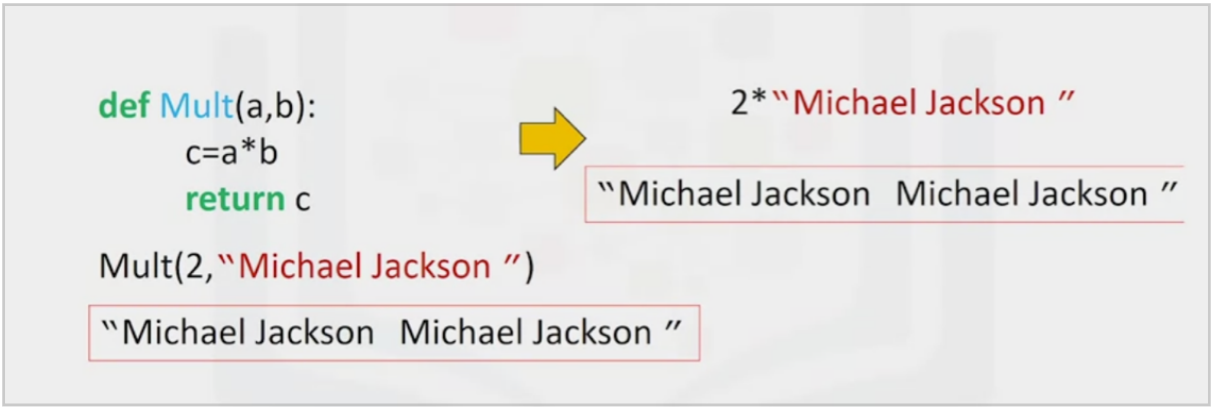
happen again with a different value of "A" The function returns a value, in this case, 9. Again, this is just a helpful analogy.



Let's try and make this function more complex. It's customary to document the function on the first few lines; this tells anyone who uses the function what it does. This documentation is surrounded in triple quotes. You can use the help command on the function to display the documentation as follows. This will printout the function name and the documentation. We will not include the documentation in the rest of the examples.



A function can have multiple parameters. The function `mult` multiplies two numbers; in other words, it finds their product. If we pass the integers 2 and 3, the result is a new integer. If we pass the integer 10 and the float 3.14, the result is a float 31.4.

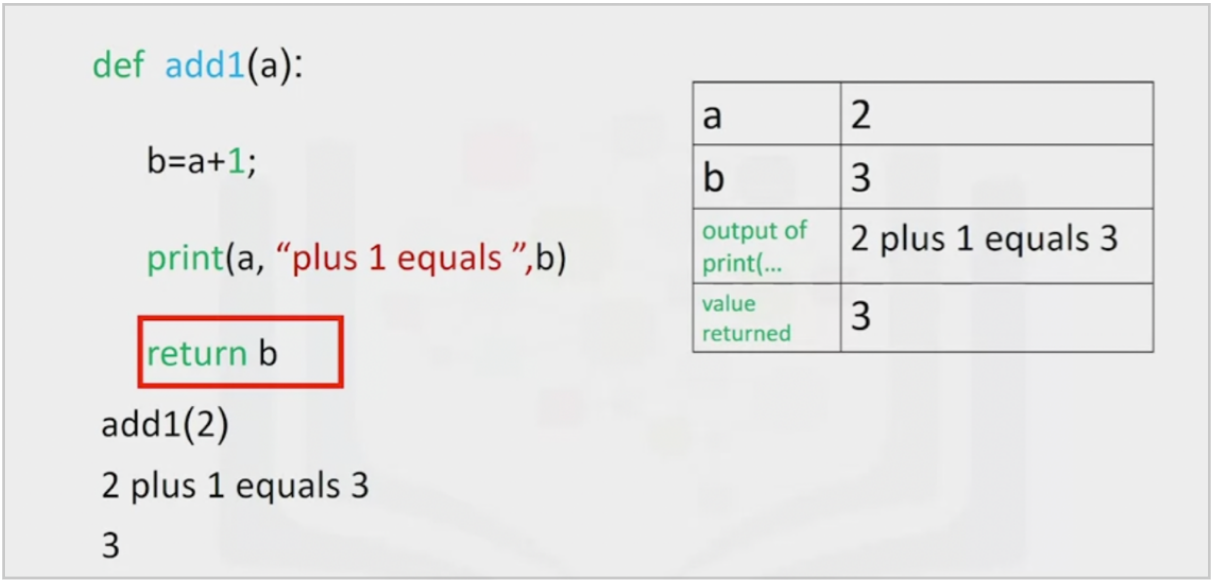


If we pass in the integer two and the string "Michael Jackson," the string Michael Jackson is repeated two times. This is because the multiplication symbol can also mean repeat a sequence. If you accidentally multiply an integer and a String instead of two integers, you won't get an error. Instead, you will get a String, and your program will progress, potentially failing later because you have a String where you expected an integer. This property will make coding simpler, but you must test your code more thoroughly.

In many cases a **function does not have a return statement**. In these cases, Python will return the special "None" object. Practically speaking, if your function has no return statement, you can treat it as if the function returns nothing at all. The function MJ simply prints the name 'Michael Jackson'. We call the function. The function prints "Michael Jackson."

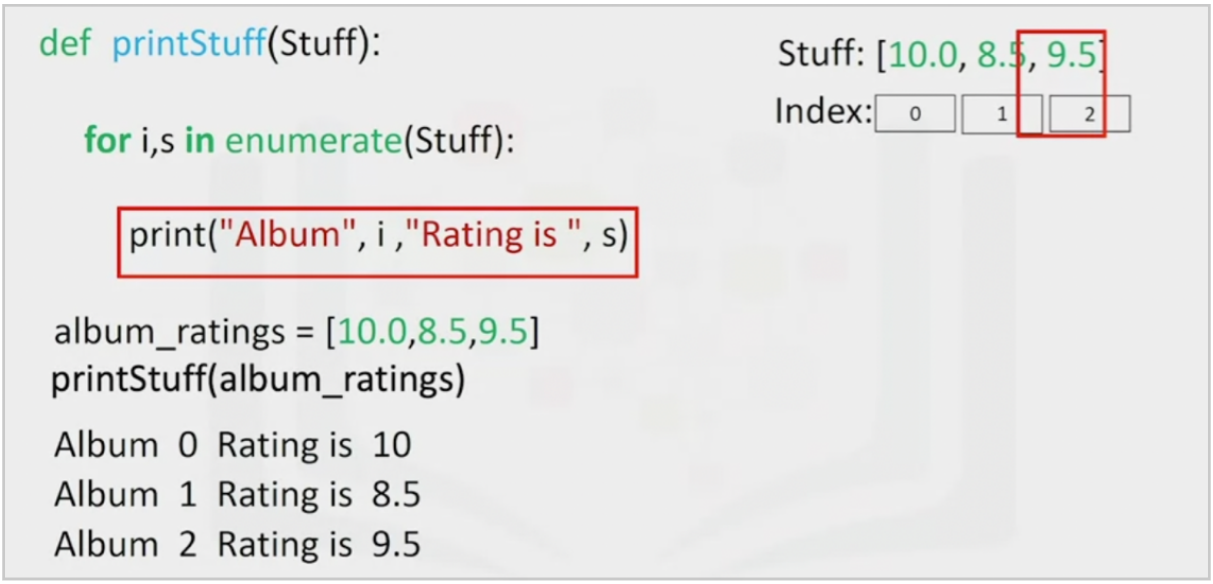
Let's define the function "No work" that performs no task. **Python doesn't allow a function to have an empty body**, so we can use the keyword pass, which doesn't do anything, but satisfies the requirement of a non-empty body. If we call the function and print it out, the function returns a `None`. In the background, if the return statement is not called, Python will automatically return a None. It is helpful to view the function No Work

with the following return statement.



Usually, functions perform more than one task. This function **prints a statement then returns a value**. Let's use this table to represent the different values as the function is called.

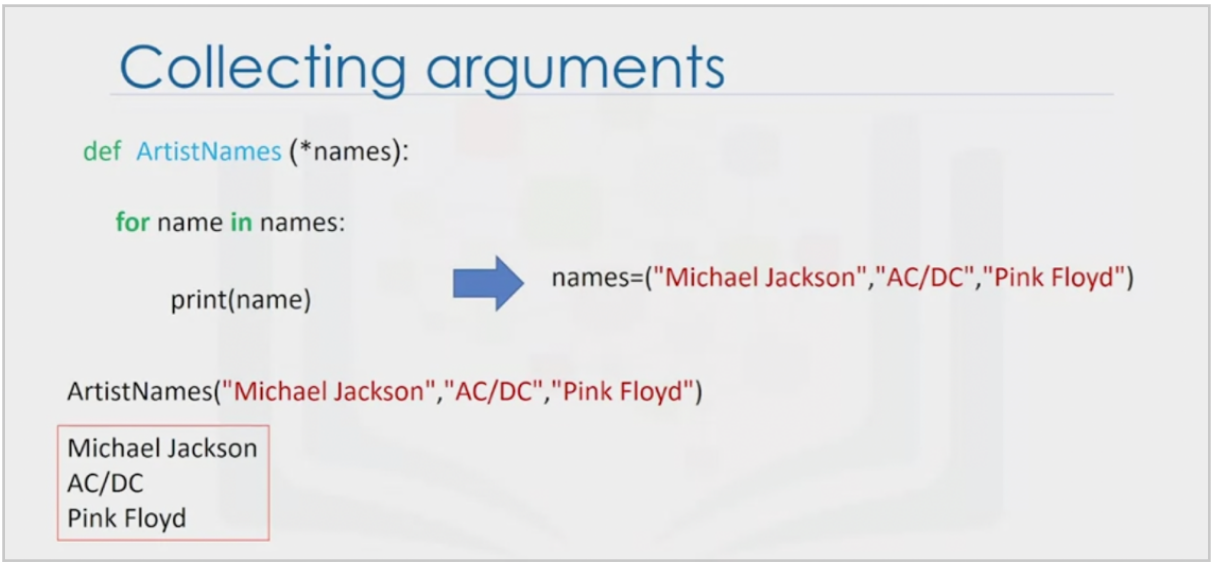
1. We call the function with an input of 2.
2. We find the value of b.
3. The function prints the statement with the values of a and b.
4. Finally, the function returns the value of b, in this case, 3.



We can use loops in functions. This function **prints out the values and indexes of a loop or tuple**. We call the function with the **list** `album ratings` as an input. Let's display the list on the right with its corresponding index.

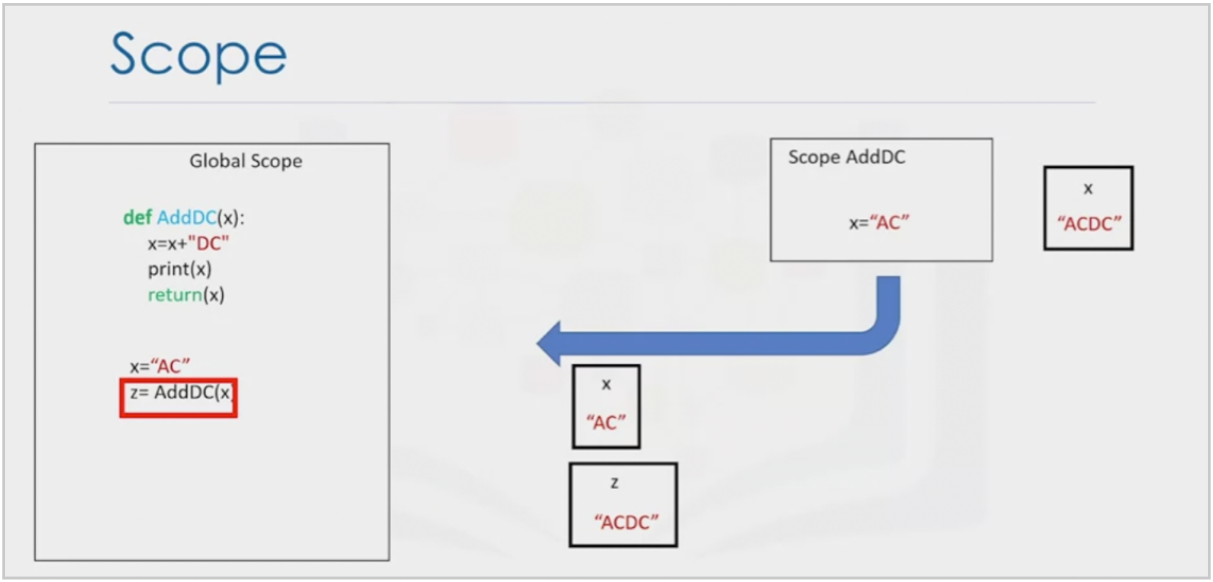
1. `Stuff` is used as an input to the function enumerate.
2. This operation will pass the index to `i` and the value in the list to `s`.
3. The function would begin to iterate through the loop.
4. The function will print the first index and the first value in the list.

5. We continue iterating through the loop. The values of I and s are updated. The print statement is reached.
6. Similarly, the next values of the list and index are printed.
7. The process is repeated. The values of I and s are updated.
8. We continue iterating until the final values in the list are printed out.



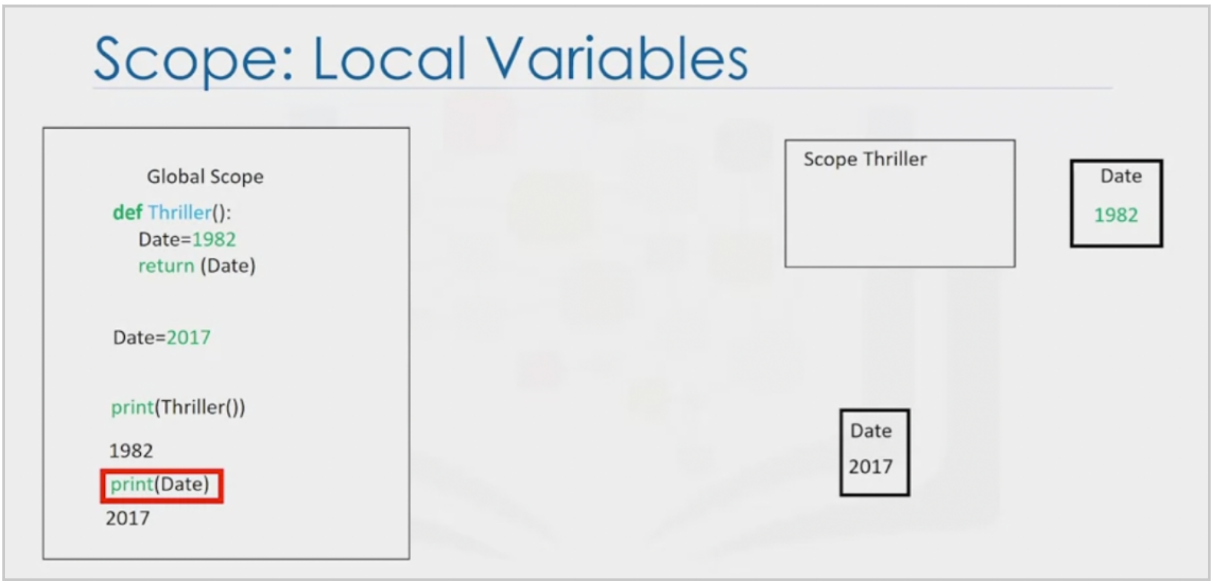
Variadic parameters allow us to **input a variable number of elements**. Consider the following function; the function has an asterisk on the parameter names. When we call the function, three parameters are packed into the tuple names. We then iterate through the loop; the values are printed out accordingly.

If we call the same function with only two parameters as inputs, the variable names only contain two elements. The result is only two values are printed out.

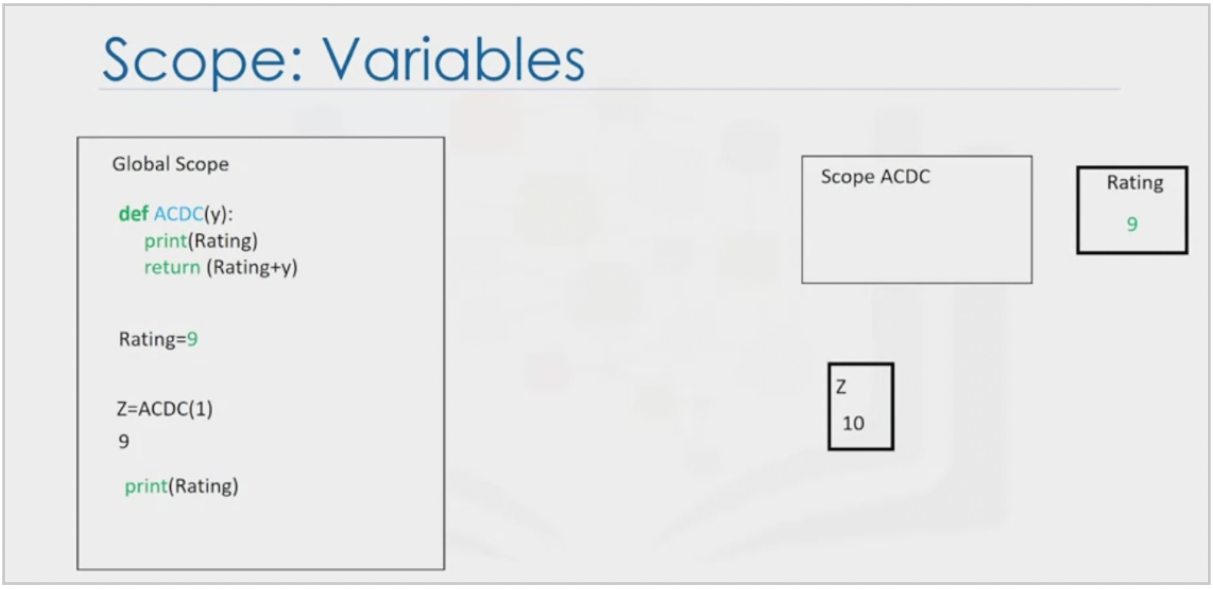


The `scope` of a variable is the part of the program where that variable is accessible. Variables that are defined outside of any function are said to be within the global scope, meaning they can be accessed anywhere after they are defined. Here we have a function

that adds the string DC to the parameter x. When we reach the part where the value of x is set to AC, this is within the global scope, meaning x is accessible anywhere after it is defined. **A variable defined in the global scope is called a global variable**. When we call the function, we enter a new scope or the scope of AddDC. We pass as an argument to the AddDC function, in this case, AC. Within the scope of the function, the value of x is set to ACDC. The function returns the value and is assigned to z. Within the global scope, the value z is set to ACDC. After the value is returned, the scope of the function is deleted.



`Local variables` only exist within the scope of a function. Consider the function thriller; the local variable Date is set to 1982. When we call the function, we create a new scope. Within that scope of the function, the value of the date is set to 1982.
The value of date does not exist within the global scope. **Variables inside the global scope can have the same name as variables in the local scope with no conflict**. Consider the function thriller; the **local variable** `Date` is set to 1982. The **global variable** `Date` is set to 2017. When we call the function, we create a new scope. Within that scope, the value of the date is set to 1982. If we call the function, it returns the value of Date in the local scope, in this case, 1982. When we print in the global scope, we use the global variable value. The global value of the variable is 2017. Therefore, the value is set to 2017.



If a variable is not defined within a function, Python will check the global scope. Consider the function "AC-DC". The function has the variable rating, with no value assigned. If we define the variable rating in the global scope, then call the function, Python will see there is no value for the variable Rating. As a result, python will leave the scope and check if the variable Ratings exists in the global scope. It will use this value of Ratings in the global scope within the scope of "AC-DC". In the function, will print out a 9. The value of z in the global scope will be 10, as we added one. The value of rating will be unchanged within the global scope.

Consider the function Pink Floyd. If we define the variable Claimed Sales with the keyword global, the variable will be a global variable. We call the function Pink Floyd. The variable claimed sales is set to the string "45 million" in the global scope. When we print the variable, we get a value of "45 million." There is a lot more you can do with functions.

Functions Lab

This notebook will teach you about the functions in the Python Programming Language. By the end of this lab, you'll know the basic concepts about function, variables, and how to use functions.

Objectives

After completing this lab you will be able to:

- Understand functions and variables
- Work with functions and variables

Table of Contents

Functions

A function is a reusable block of code which performs operations specified in the function. They let you break down tasks and allow you to reuse your code in different programs.
There are two types of functions :

- **Pre-defined functions**
- **User defined functions**

What is a Function?

You can define functions to provide the required functionality. Here are simple rules to define a function in Python:

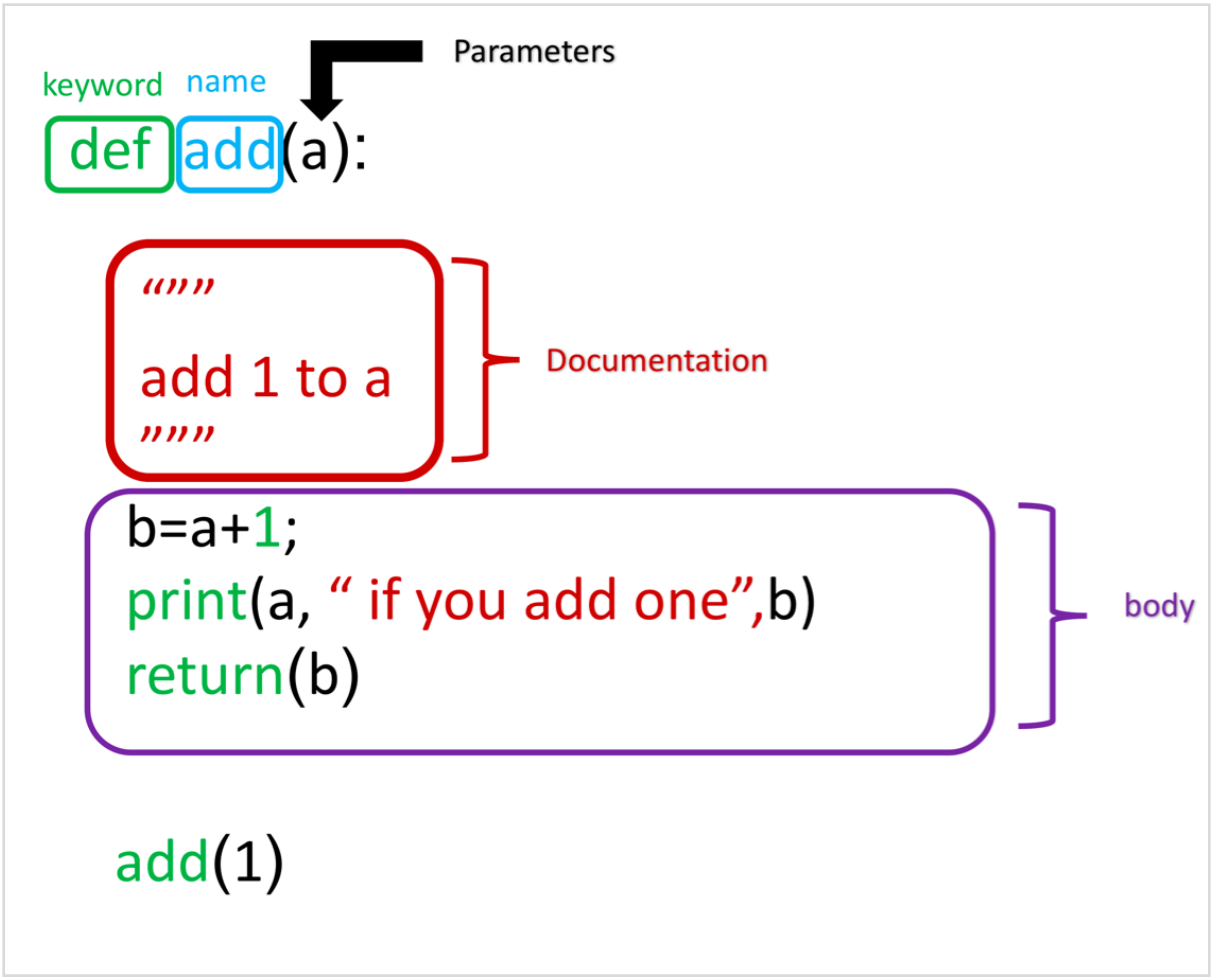
- Functions blocks begin `def` followed by the function `name` and parentheses `()`.
- There are input parameters or arguments that should be placed within these parentheses.
- You can also define parameters inside these parentheses.
- There is a body within every function that starts with a colon (`:`) and is indented.
- You can also place documentation before the body.

- The statement `return` exits a function, optionally passing back a value.

An example of a function that adds on to the parameter `a` prints and returns the output as `b`:

```
# First function example: Add 1 to a and store as b

def add(a):
    """
    add 1 to a
    """
    b = a + 1
    print(a, "if you add one", b)
    return(b)
```

The figure below illustrates the terminology:



We can obtain help about a function :

```
# Get a help on add function

help(add)
```

```
    Help on function add in module __main__:

    add(a)
        add 1 to a
```

We can call the function:

```
# Call the function add()

add(1)
```

```
    1 if you add one 2
    2
```

If we call the function with a new input we get a new result:

```
# Call the function add()

add(2)
```

```
    2 if you add one 3
    3
```

We can create different functions. For example, we can create a function that multiplies two numbers. The numbers will be represented by the variables `a` and `b`:

```
# Define a function for multiple two numbers

def Mult(a, b):
    c = a * b
    return(c)
    print('This is not printed')

result = Mult(12,2)
print(result)
```

```
24
```

The same function can be used for different data types. For example, we can multiply two integers:

```
# Use mult() multiply two integers

Mult(2, 3)
```

```
6
```

Note how the function terminates at the `return` statement, while passing back a value. This value can be further assigned to a different variable as desired.

The same function can be used for different data types. For example, we can multiply two integers:

Two Floats:

```
# Use mult() multiply two floats
```

```
Mult(10.0, 3.14)
```

31.400000000000002

We can even replicate a string by multiplying with an integer:

```
# Use mult() multiply two different type values together

Mult(2, "Michael Jackson ")
```

'Michael Jackson Michael Jackson '

Variables

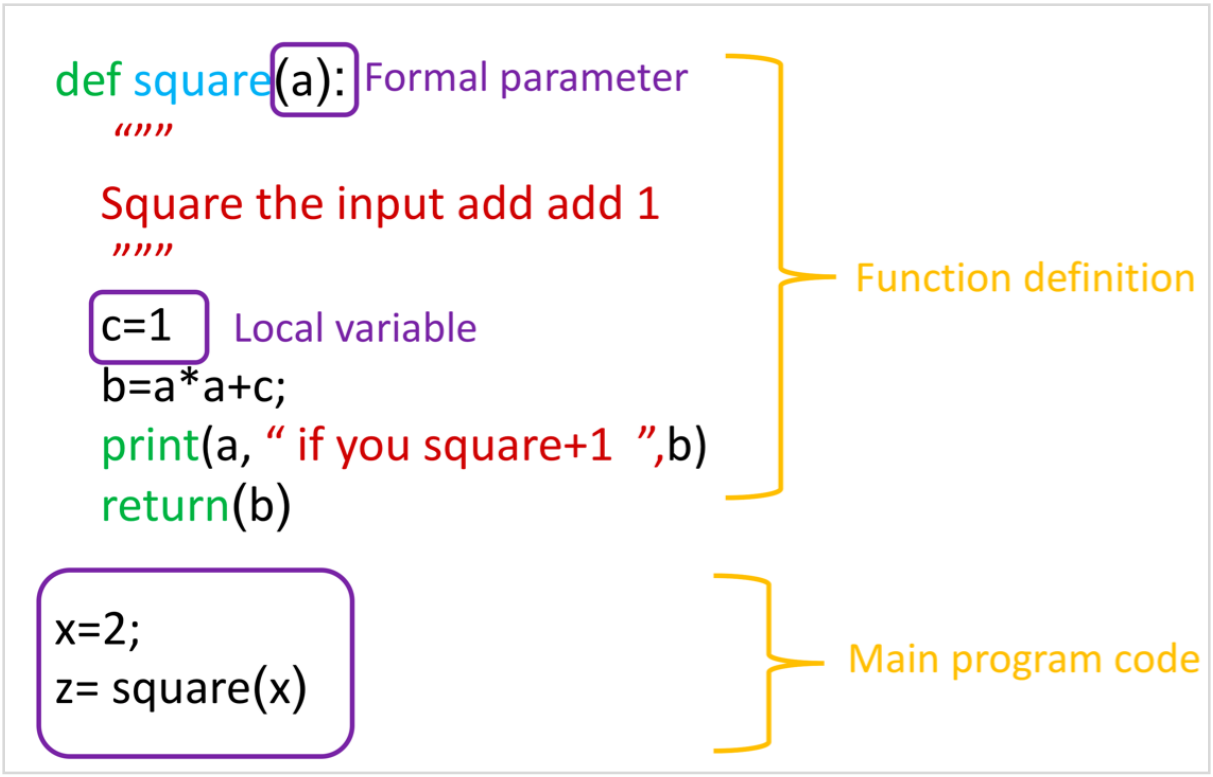
The input to a function is called a formal parameter.

A variable that is declared inside a function is called a local variable. The parameter only exists within the function (i.e. the point where the function starts and stops).

A variable that is declared outside a function definition is a global variable, and its value is accessible and modifiable throughout the program. We will discuss more about global variables at the end of the lab.

```
# Function Definition

def square(a):
    # Local variable b
    b = 1
    c = a * a + b
    print(a, "if you square + 1", c)
    return(c)
```

The labels are displayed in the figure:



We can call the function with an input of **3**:

```
# Initializes Global variable

x = 3
# Makes function call and return function a y
y = square(x)
y
```

```
    3 if you square + 1 10
    10
```

We can call the function with an input of **2** in a different manner:

```
# Directly enter a number as parameter
square(2)
```

```
    2 if you square + 1 5
    5
```

If there is no `return` statement, the function returns `None`. The following two functions are equivalent:

```
# Define functions, one with return value None and other without return value
def MJ():
    print('Michael Jackson')

def MJ1():
    print('Michael Jackson')
    return(None)
```

```
# See the output
MJ()
```

Michael Jackson

```
# See the output
MJ1()
```

Michael Jackson

Printing the function after a call reveals a **None** is the default return statement:

```
# See what functions returns are
print(MJ())
print(MJ1())
```

```
    Michael Jackson
    None
    Michael Jackson
    None
```

Create a function `con` that concatenates two strings using the addition operation:

```
# Define the function for combining strings
def con(a, b):
    return(a + b)


# Test on the con() function

con("This ", "is")
```

'This is'

[Tip] How do I learn more about the pre-defined functions in Python?

We will be introducing a variety of pre-defined functions to you as you learn more about Python. There are just too many functions, so there's no way we can teach them all in one sitting. But if you'd like to take a quick peek, here's a short reference card for some of the commonly-used pre-defined functions: Reference

---

Functions Make Things Simple

Consider the two lines of code in **Block 1** and **Block 2**: the procedure for each block is identical. The only thing that is different is the variable names and values.

Block 1:

```
# a and b calculation block1

a1 = 4
b1 = 5
```

```
c1 = a1 + b1 + 2 * a1 * b1 - 1
if(c1 < 0):
    c1 = 0
else:
    c1 = 5
c1
```

5

Block 2:

```
# a and b calculation block2

a2 = 0
b2 = 0
c2 = a2 + b2 + 2 * a2 * b2 - 1
if(c2 < 0):
    c2 = 0
else:
    c2 = 5
c2
```

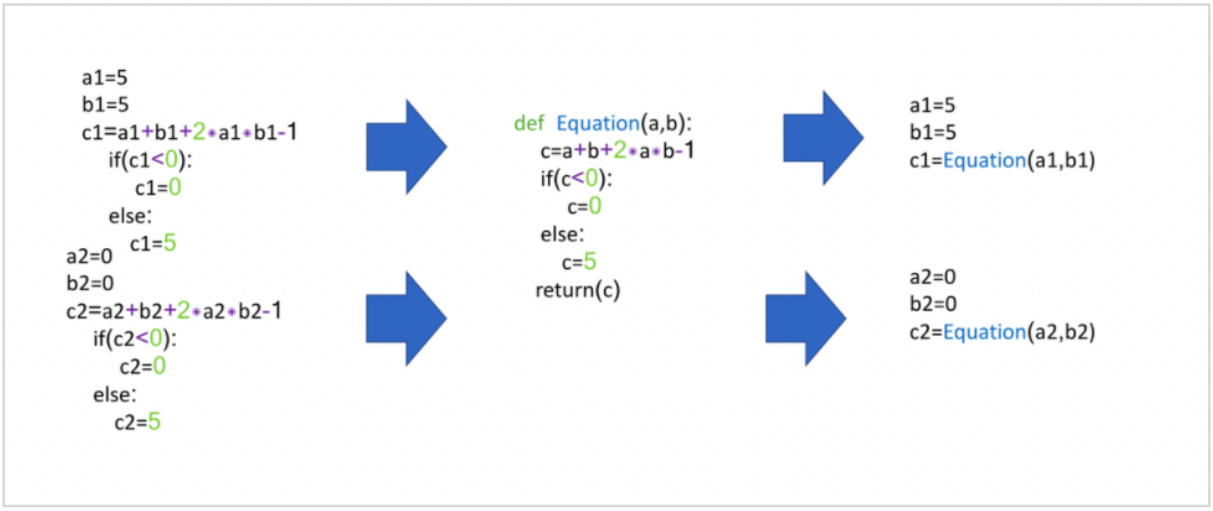
0

We can replace the lines of code with a function. A function combines many instructions into a single line of code. Once a function is defined, it can be used repeatedly. You can invoke the same function many times in your program. You can save your function and use it in another program or use someone else's function. The lines of code in code **Block 1** and code **Block 2** can be replaced by the following function:

```
# Make a Function for the calculation above

def Equation(a,b):
    c = a + b + 2 * a * b - 1
    if(c < 0):
        c = 0
```

```
    else:
        c = 5
    return(c)
```

This function takes two inputs, a and b, then applies several operations to return c.
We simply define the function, replace the instructions with the function, and input the new values of `a1`, `b1` and `a2`, `b2` as inputs. The entire process is demonstrated in the figure:



Code **Blocks 1** and **Block 2** can now be replaced with code **Block 3** and code **Block 4**.

Block 3:

```
a1 = 4
b1 = 5
c1 = Equation(a1, b1)
c1
```

```
5
```

Block 4:

```
a2 = 0
b2 = 0
c2 = Equation(a2, b2)
c2
```

```

```

0

Pre-defined functions

There are many pre-defined functions in Python, so let's start with the simple ones.
The `print()` function:

```
# Build-in function print()

album_ratings = [10.0, 8.5, 9.5, 7.0, 7.0, 9.5, 9.0, 9.5]
print(album_ratings)
```

[10.0, 8.5, 9.5, 7.0, 7.0, 9.5, 9.0, 9.5]

The `sum()` function adds all the elements in a list or tuple:

```
# Use sum() to add every element in a list or tuple together

sum(album_ratings)
```

70.0

The `len()` function returns the length of a list or tuple:

```
# Show the length of the list or tuple

len(album_ratings)
```

8

Using `if` / `else` Statements and Loops in Functions

The `return()` function is particularly useful if you have any IF statements in the function,

when you want your output to be dependent on some condition:

```
# Function example

def type_of_album(artist, album, year_released):

    print(artist, album, year_released)
    if year_released > 1980:
        return "Modern"
    else:
        return "Oldie"

x = type_of_album("Michael Jackson", "Thriller", 1980)
print(x)
```

Michael Jackson Thriller 1980
Oldie

We can use a loop in a function. For example, we can `print` out each element in a list:

```
# Print the list using for loop

def PrintList(the_list):
    for element in the_list:
        print(element)
```

```
# Implement the printlist function

PrintList(['1', 1, 'the man', "abc"])
```

```
    1
    1
    the man
```

```
    abc
```

Setting default argument values in your custom functions

You can set a default value for arguments in your function. For example, in the `isGoodRating()` function, what if we wanted to create a threshold for what we consider to be a good rating? Perhaps by default, we should have a default rating of 4:

```
# Example for setting param with default value

def isGoodRating(rating=4):
    if(rating < 7):
        print("this album sucks it's rating is",rating)

    else:
        print("this album is good its rating is",rating)
```

```
# Test the value with default value and with input

isGoodRating()
isGoodRating(10)
```

```
    this album sucks it's rating is 4
    this album is good its rating is 10
```

Global variables

So far, we've been creating variables within functions, but we have not discussed variables outside the function. These are called global variables. Let's try to see what `printer1` returns:

```
```

```
# Example of global variable

artist = "Michael Jackson"
def printer1(artist):
    internal_var1 = artist
    print(artist, "is an artist")

printer1(artist)
# try runningthe following code
printer1(internal_var1)
```

```
Michael Jackson is an artist
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_4751/1916328705.py in <module>
      8 printer1(artist)
      9 # try runningthe following code
---> 10 printer1(internal_var1)

NameError: name 'internal_var1' is not defined
```

We got a Name Error: `name 'internal_var' is not defined`. **Why?**
It's because all the variables we create in the function is a **local variable**, meaning that the variable assignment does not persist outside the function.

But there is a way to create **global variables** from within a function as follows:

```
artist = "Michael Jackson"

def printer(artist):
    global internal_var
    internal_var= "Whitney Houston"
    print(artist,"is an artist")

printer(artist)
```

```
printer(internal_var)
```

Michael Jackson is an artist
Whitney Houston is an artist

Scope of a Variable

The scope of a variable is the part of that program where that variable is accessible. Variables that are declared outside of all function definitions, such as the `myFavouriteBand` variable in the code shown here, are accessible from anywhere within the program. As a result, such variables are said to have global scope, and are known as global variables. `myFavouriteBand` is a global variable, so it is accessible from within the `getBandRating` function, and we can use it to determine a band's rating. We can also use it outside of the function, such as when we pass it to the print function to display it:

```
# Example of global variable

myFavouriteBand = "AC/DC"

def getBandRating(bandname):
    if bandname == myFavouriteBand:
        return 10.0
    else:
        return 0.0

print("AC/DC's rating is:", getBandRating("AC/DC"))
print("Deep Purple's rating is:",getBandRating("Deep Purple"))
print("My favourite band is:", myFavouriteBand)



    AC/DC's rating is: 10.0
    Deep Purple's rating is: 0.0
    My favourite band is: AC/DC
```

Take a look at this modified version of our code. Now the `myFavouriteBand` variable is defined within the `getBandRating` function. A variable that is defined within a function is said to be a local variable of that function. That means that it is only accessible from within

the function in which it is defined. Our `getBandRating` function will still work, because `myFavouriteBand` is still defined within the function. However, we can no longer print `myFavouriteBand` outside our function, because it is a local variable of our `getBandRating` function; it is only defined within the `getBandRating` function:

```
# Deleting the variable "myFavouriteBand" from the previous example to
demonstrate an example of a local variable

del myFavouriteBand

# Example of local variable

def getBandRating(bandname):
    myFavouriteBand = "AC/DC"
    if bandname == myFavouriteBand:
        return 10.0
    else:
        return 0.0

print("AC/DC's rating is: ", getBandRating("AC/DC"))
print("Deep Purple's rating is: ", getBandRating("Deep Purple"))
print("My favourite band is", myFavouriteBand)
```

```
AC/DC's rating is:  10.0
Deep Purple's rating is:  0.0
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_4751/418399712.py in <module>
      15 print("AC/DC's rating is: ", getBandRating("AC/DC"))
      16 print("Deep Purple's rating is: ", getBandRating("Deep Purple"))
---> 17 print("My favourite band is", myFavouriteBand)

NameError: name 'myFavouriteBand' is not defined
```

Finally, take a look at this example. We now have two `myFavouriteBand` variable definitions.

The first one of these has a global scope, and the second of them is a local variable within the `getBandRating` function. Within the `getBandRating` function, the local variable takes precedence. **Deep Purple** will receive a rating of 10.0 when passed to the `getBandRating` function. However, outside of the `getBandRating` function, the `getBandRating` s local variable is not defined, so the `myFavouriteBand` variable we print is the global variable, which has a value of **AC/DC**:

```
# Example of global variable and local variable with the same name

myFavouriteBand = "AC/DC"

def getBandRating(bandname):
    myFavouriteBand = "Deep Purple"
    if bandname == myFavouriteBand:
        return 10.0
    else:
        return 0.0

print("AC/DC's rating is:",getBandRating("AC/DC"))
print("Deep Purple's rating is: ",getBandRating("Deep Purple"))
print("My favourite band is:",myFavouriteBand)
```

```
    AC/DC's rating is: 0.0
    Deep Purple's rating is:  10.0
    My favourite band is: AC/DC
```

Collections and Functions

When the number of arguments are unknown for a function, They can all be packed into a tuple as shown:

```
def printAll(*args): # All the arguments are 'packed' into args which can be
treated like a tuple
    print("No of arguments:", len(args))
    for argument in args:
```

```
        print(argument)
#printAll with 3 arguments
printAll('Horsefeather','Adonis','Bone')
#printAll with 4 arguments
printAll('Sidecar','Long Island','Mudslide','Carriage')
```

```
    No of arguments: 3
    Horsefeather
    Adonis
    Bone
    No of arguments: 4
    Sidecar
    Long Island
    Mudslide
    Carriage
```

Similarly, The arguments can also be packed into a dictionary as shown:

```
def printDictionary(**args):
    for key in args:
        print(key + " : " + args[key])

printDictionary(Country='Canada',Province='Ontario',City='Toronto')
```

```
    Country : Canada
    Province : Ontario
    City : Toronto
```

Functions can be incredibly powerful and versatile. They can accept (and return) data types, objects and even other functions as arguements. Consider the example below:

```
def addItems(list):
```

```
    list.append("Three")
    list.append("Four")

myList = ["One","Two"]
addItems(myList)
myList
```

['One', 'Two', 'Three', 'Four']

Note how the changes made to the list are not limited to the functions scope. This occurs as it is the lists **reference** that is passed to the function - Any changes made are on the orignal instance of the list. Therefore, one should be cautious when passing mutable objects into functions.

Quiz on Functions

Come up with a function that divides the first input by the second input:

```
# Write your code below and press Shift+Enter to execute
def devide(a,b):
    c = a/b
    return(c)

devide(3,2)
```

1.5

Use the function `con` for the following question.

```
# Use the con function for the following question

def con(a, b):
```

```
    return(a + b)
```

Can the `con` function we defined before be used to add two integers or strings?

```
# Write your code below and press Shift+Enter to execute
# yes
con(2,3)
```

5

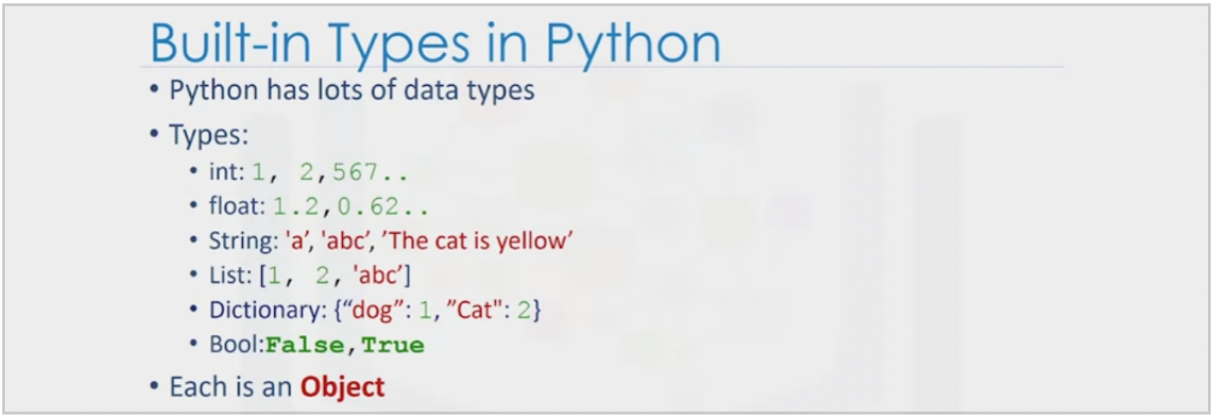
Can the `con` function we defined before be used to concatenate lists or tuples?

```
# Write your code below and press Shift+Enter to execute
# yes
con(['cane','gatto'],['pane','paste'])
```

['cane', 'gatto', 'pane', 'paste']

Object and Classes theory

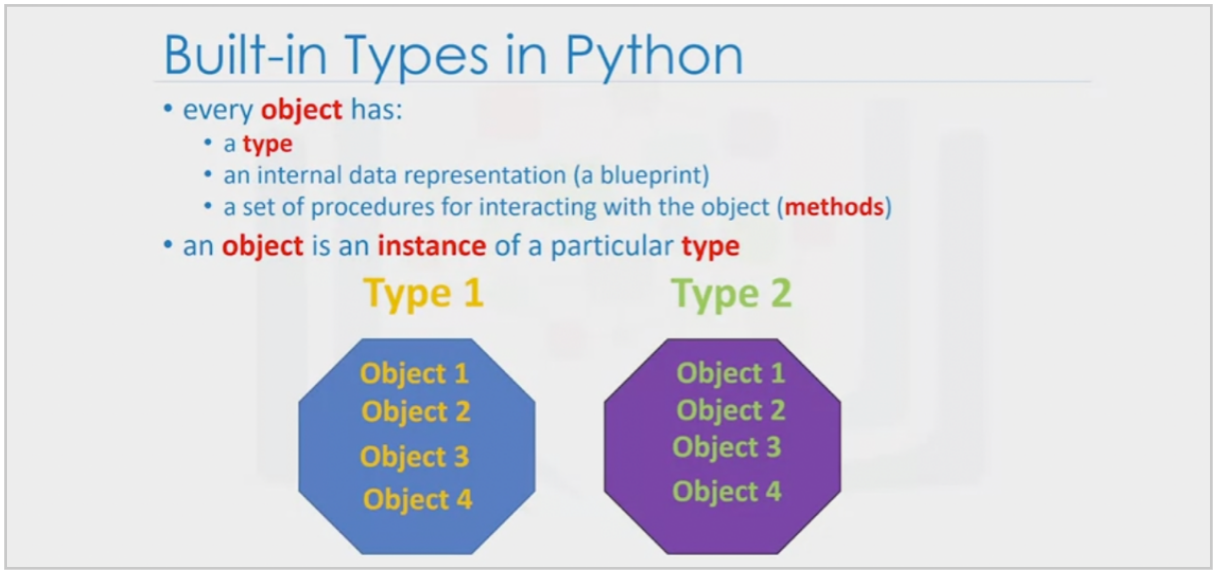
In this module, we're going to talk about objects and classes. Python has many different kinds of data types: integers, floats, strings, lists, dictionaries, booleans. In Python, each is an object.



Every object has the following:

- a **type**,

- internal representation,
- a set of functions called **methods** to interact with the data.
- An object is an instance of a particular type.

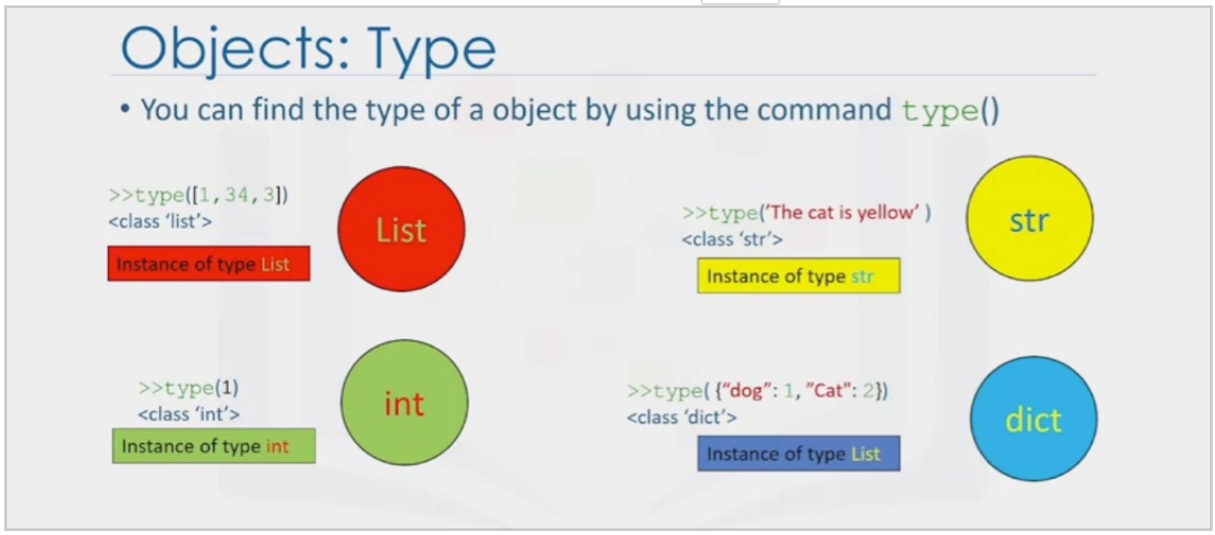


For example, we have two types, type one and type two. We can have several objects of type one as shown in yellow. Each object is an instance of type one. We also have several objects of type two shown in green. Each object is an instance of type two. Let's do several less abstract examples.

Every time we create an integer, we are creating an instance of type integer, or we are creating an integer object. In this case, we are creating five instances of type integer or five integer objects.

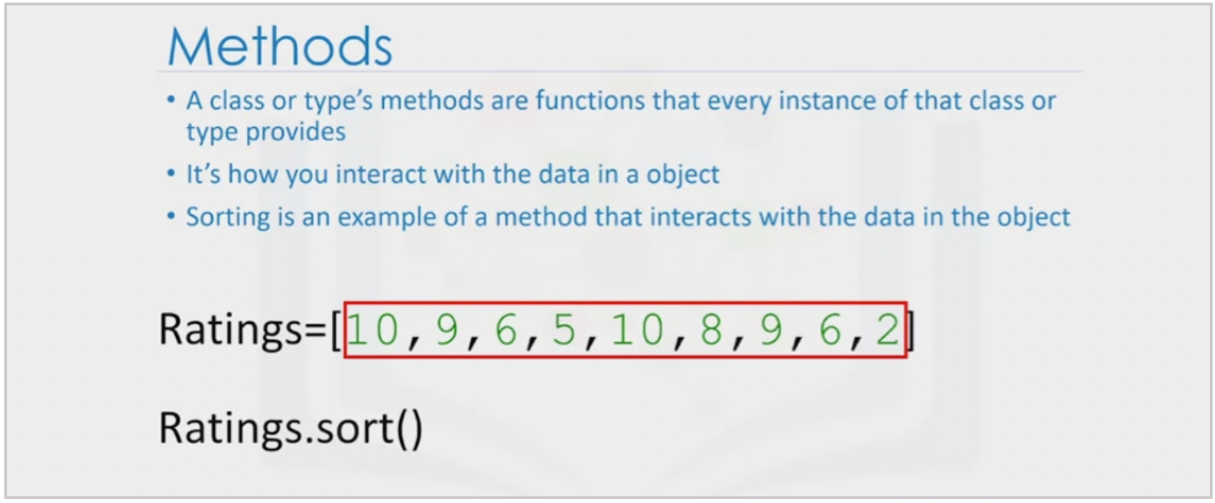
Similarly, every time we create a list, we are creating an instance of type list, or we are creating a list object. In this case, we are creating five instances of type list or five list objects.

We could find out the type of an object by using the `type` command.

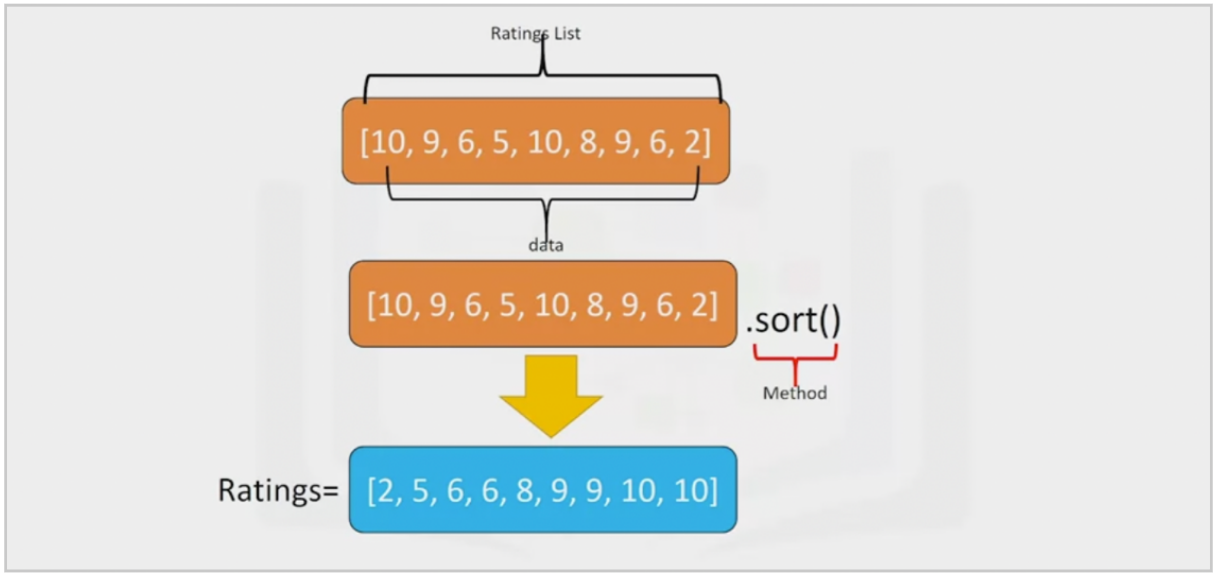


In this case, we have an object of type list, we have an object of type integer, we have an object of type string. Finally, we have an object of type dictionary.

A class or type's `methods` are functions that **every instance of that class or type provides**. It's how you interact with the object. We have been using methods all this time, for example, on lists. `Sorting` is an example of a method that interacts with the data in the object.



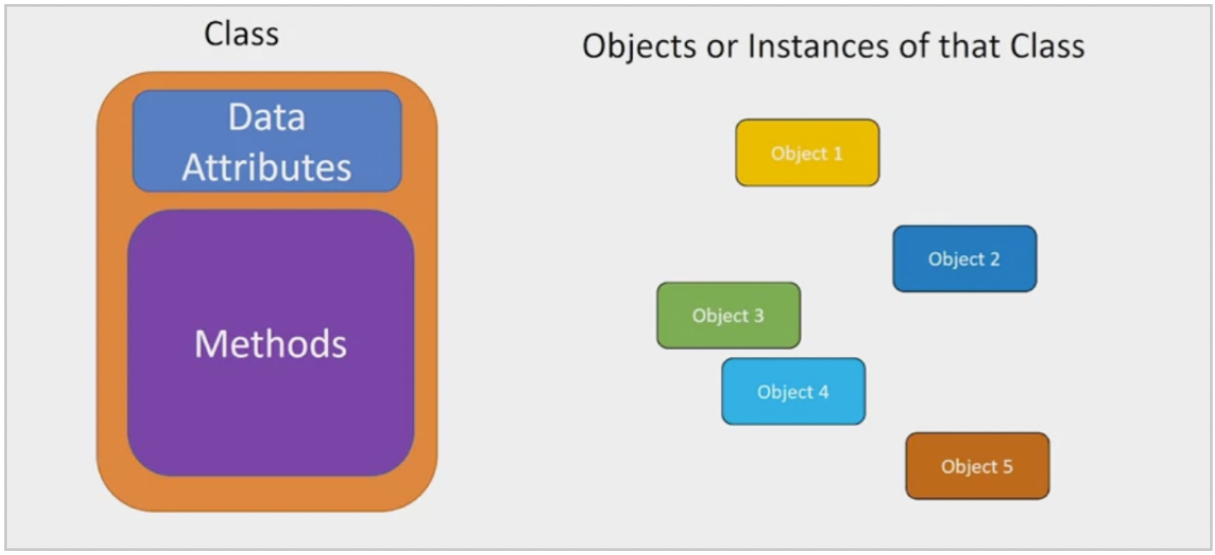
Consider the list ratings, the data is a series of numbers contained within the list. The method sort will change the data within the object. We call the method by adding a period at the end of the object's name, and the method's name we would like to call with parentheses.



We have the rating's list represented in orange. The data contained in the list is a sequence of numbers. We call the `sort method`, this **changes the data contained in the object**. You can say it changes the state of the object. We can call the `reverse` method on the list, changing the list again. We call the method, reversing the order of the sequence within the object. **In many cases, you don't have to know the inner workings of the class and its methods, you just have to know how to use them**.

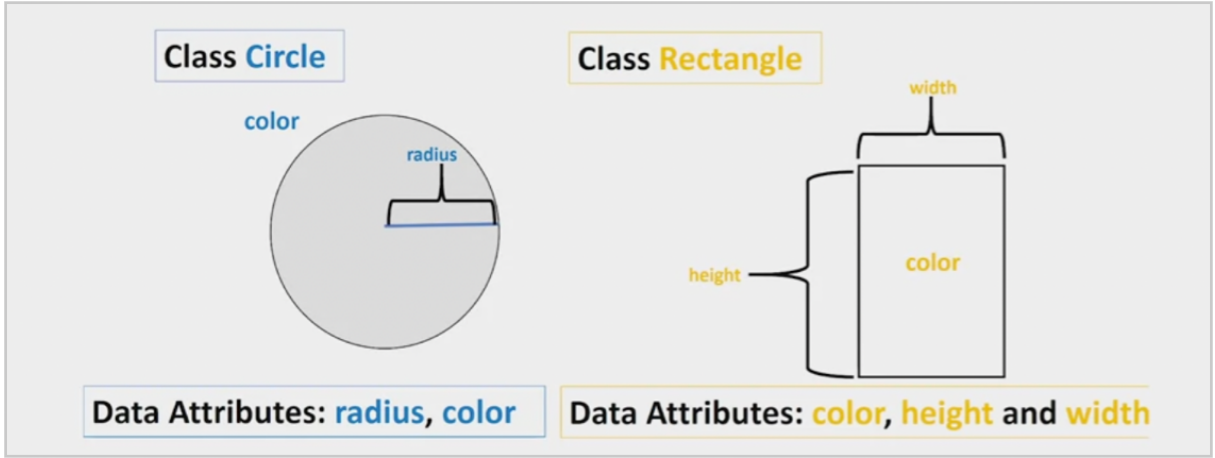
Next, we will cover how to construct your own **classes**. You can create your own type or class in Python. The class has:

- data attributes,
- methods.
- We then create instances or instances of that class or objects. The class data attributes define the class.

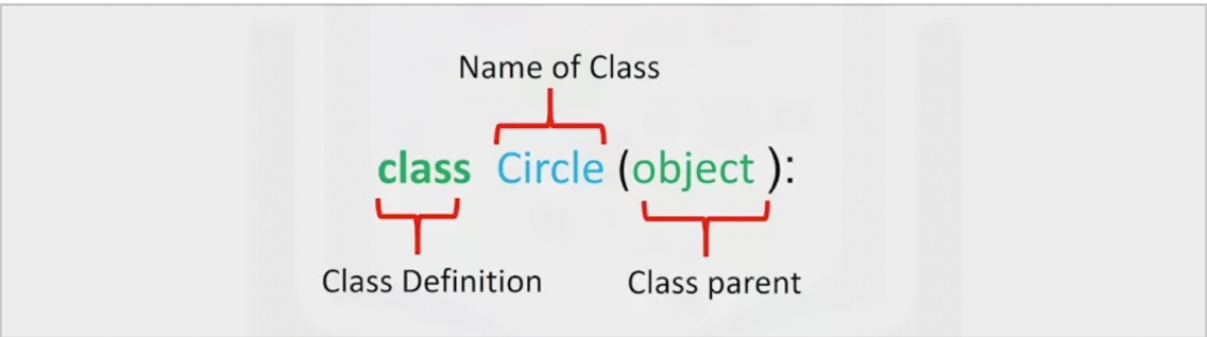


Let's create two classes.

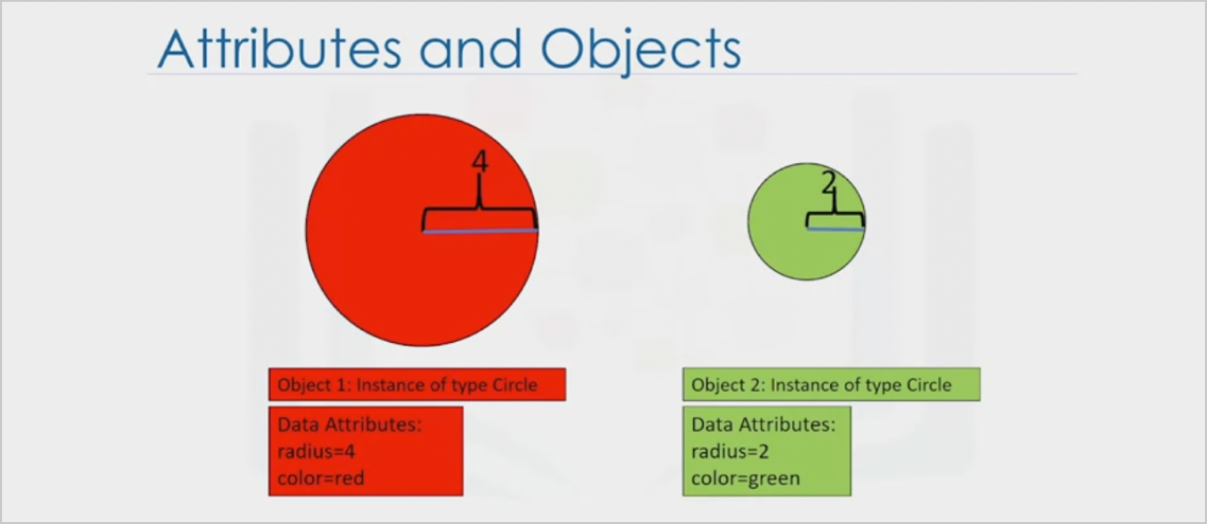
- The first class will be a circle. Let's think about what constitutes a circle. Examining this image, all we need is a **radius** to define a circle, and let's add color to make it easier to distinguish between different instances of the class later. Therefore, **our class data attributes are radius and color**.
- The second class will be rectangle. Examining the image in order to define a rectangle, we need the **height** and **width**. We will also add color to distinguish between instances later. Therefore, **the data attributes are color, height, and width**.



To create the class circle, you will need to include the class definition. This tells Python you're creating your own class, the name of the class. For this course in parentheses, you will always place the term object, this is the parent of the class.

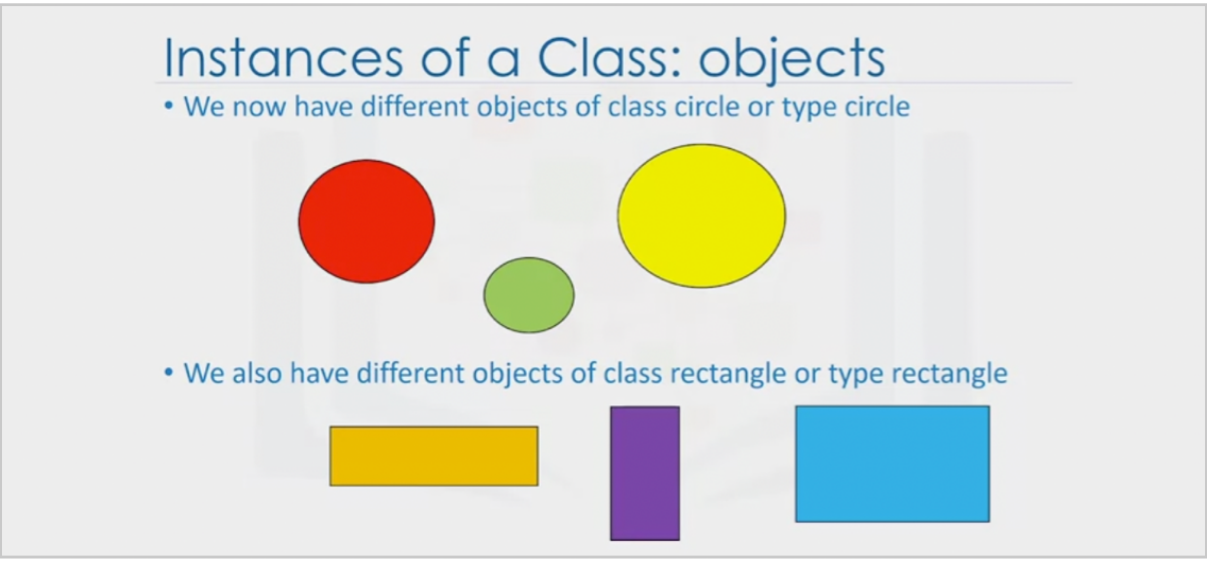


For the class rectangle, we changed the name of the class, but the rest is kept the same. Classes are outlines we have to set the attributes to create objects.



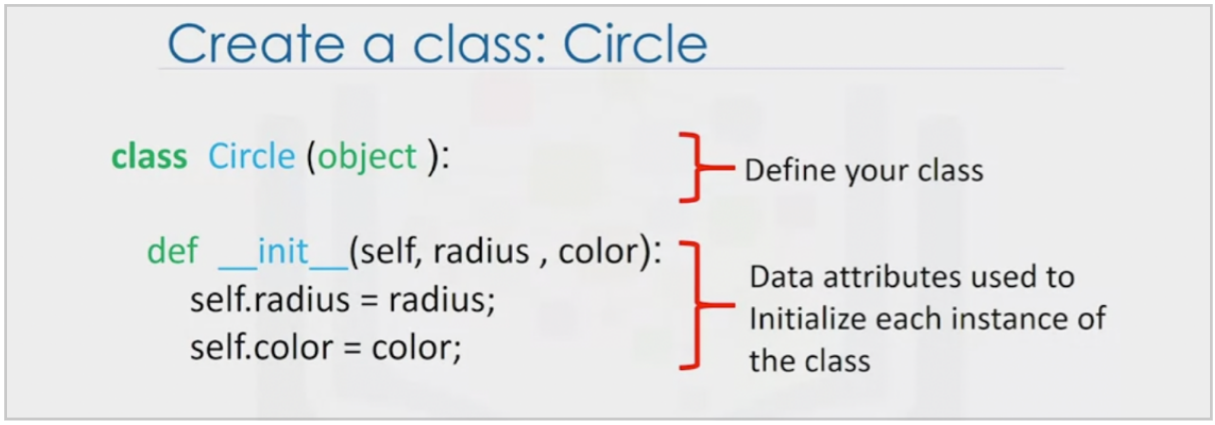
We can create an object that is an instance of type circle. The color data attribute is red, and the data attribute radius is four. We could also create a second object that is an instance of type circle. In this case, the color data attribute is green, and the data attribute radius is two.

We can also create an object that is an instance of type rectangle. The color data attribute is blue, and the data attribute of height and width is two. The second object is also an instance of type rectangle. In this case, the color data attribute is yellow, and the height is one, and the width is three.



We now have different objects of class circle or type circle. We also have different objects of class rectangle or type rectangle.

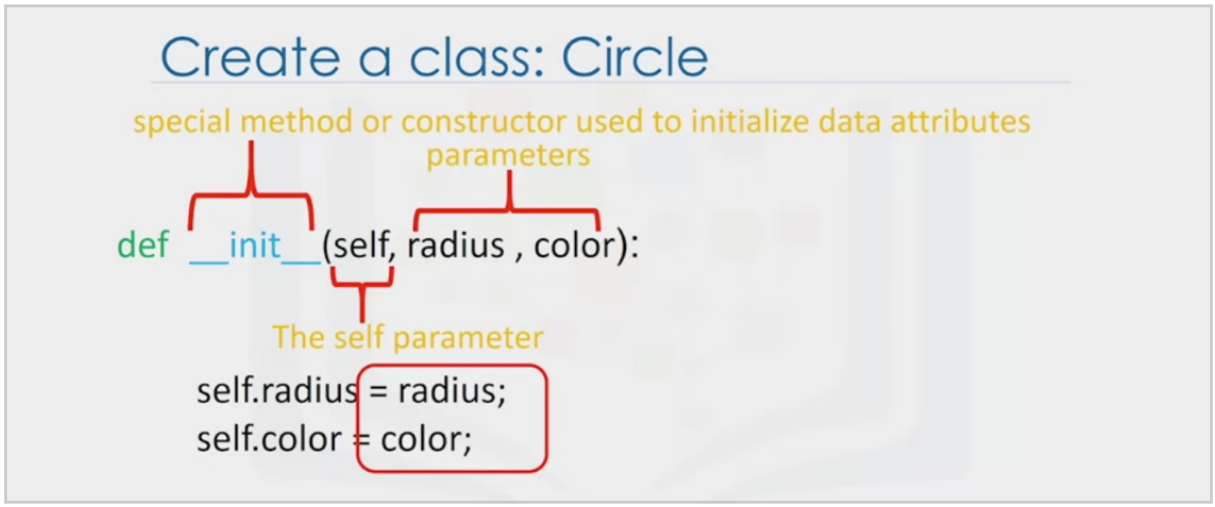
Let us continue building the circle class in Python. We define our class. We then initialise each instance of the class with data attributes, radius, and color using the class constructor.



The function `init` is a constructor. It's a special function that tells Python you are making a new class. There are other special functions in Python to make more complex classes.

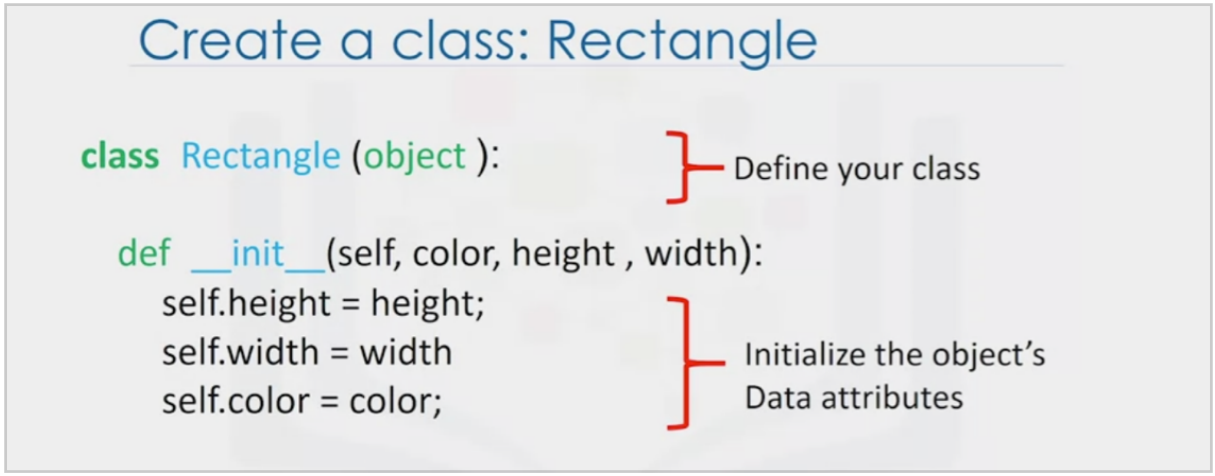
The **radius** and **color** parameters are used to initialise the radius and color data attributes of the class instance. The **self** parameter refers to the newly created instance of the class.

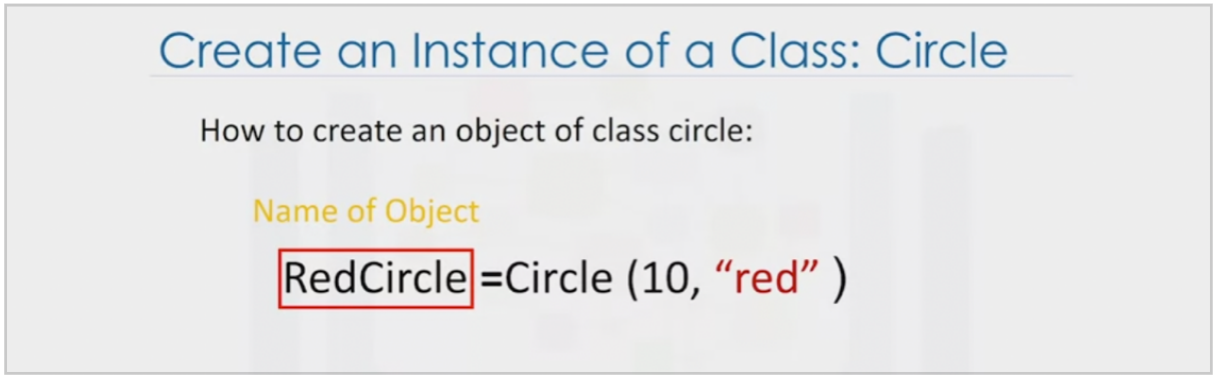
The parameters, **radius**, and **color** c an be used in the constructors body to access the values passed to the class constructor when the class is constructed.



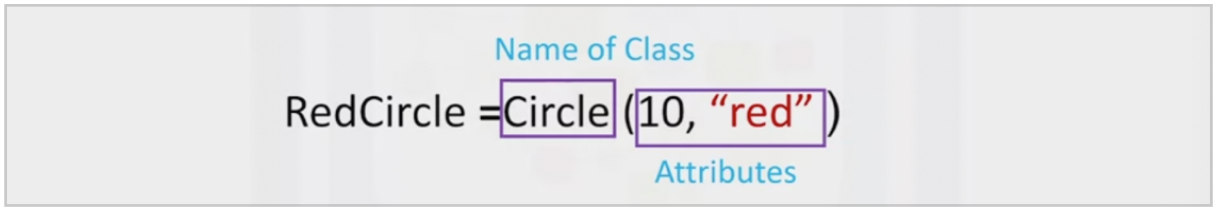
We could set the value of the radius and color data attributes to the values passed to the constructor method.

Similarly, we can define the class rectangle in Python. The name of the class is different. This time, the class data attributes are color, height, and width.



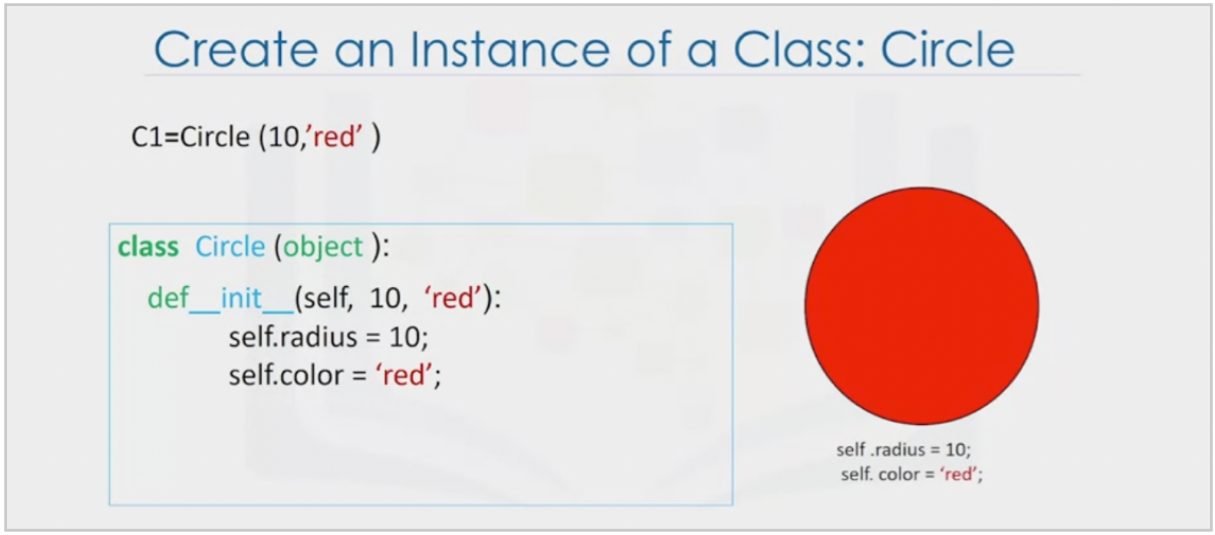


After we've created the class, in order to create an object of class circle, we introduce a **variable**. This will be the name of the object. We create the object by using the **object constructor**.

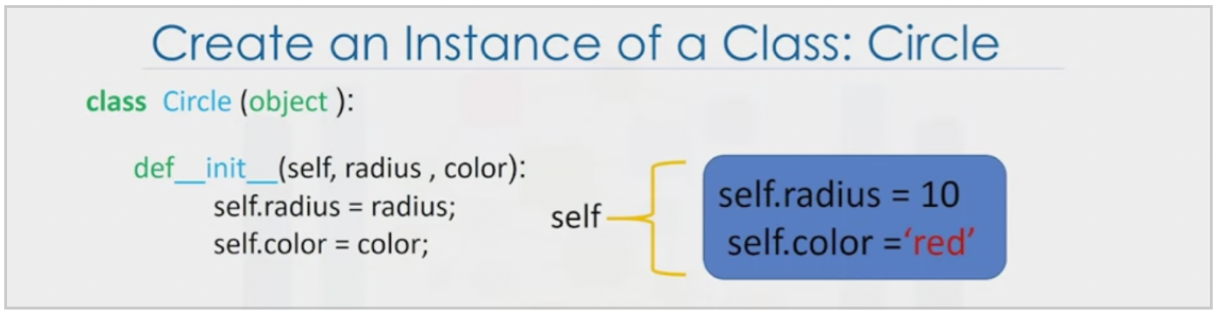


The object constructor consists of the **name of the class** as well as the **parameters**. These are the **data attributes**.

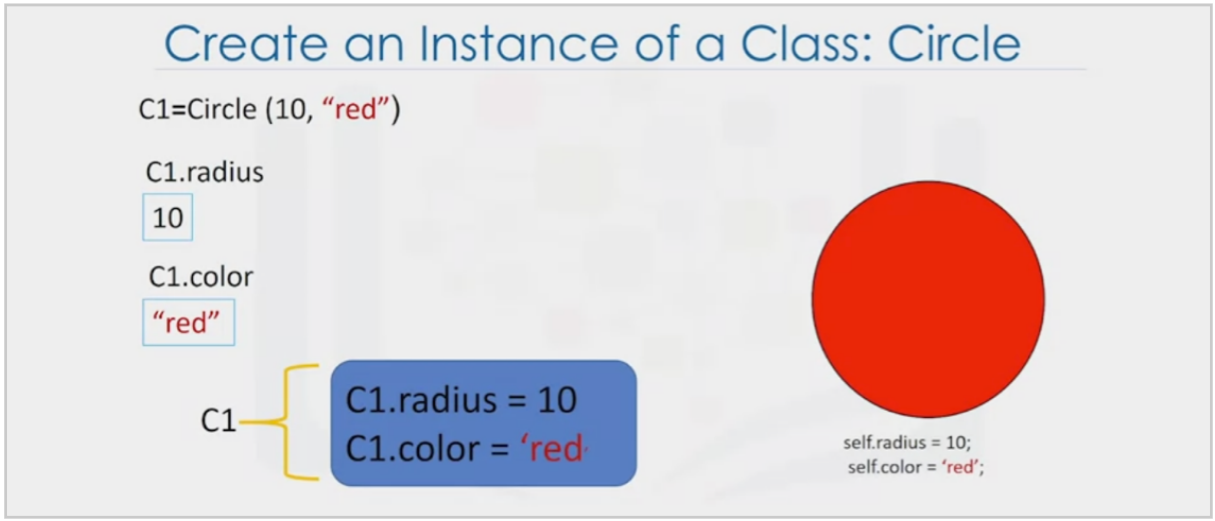
When we create a circle object, we call the code like a function. The arguments passed to the circle constructor are used to initialize the data attributes of the newly created circle instance.



It is helpful to think of **self** as a box that contains all the data attributes of the object.

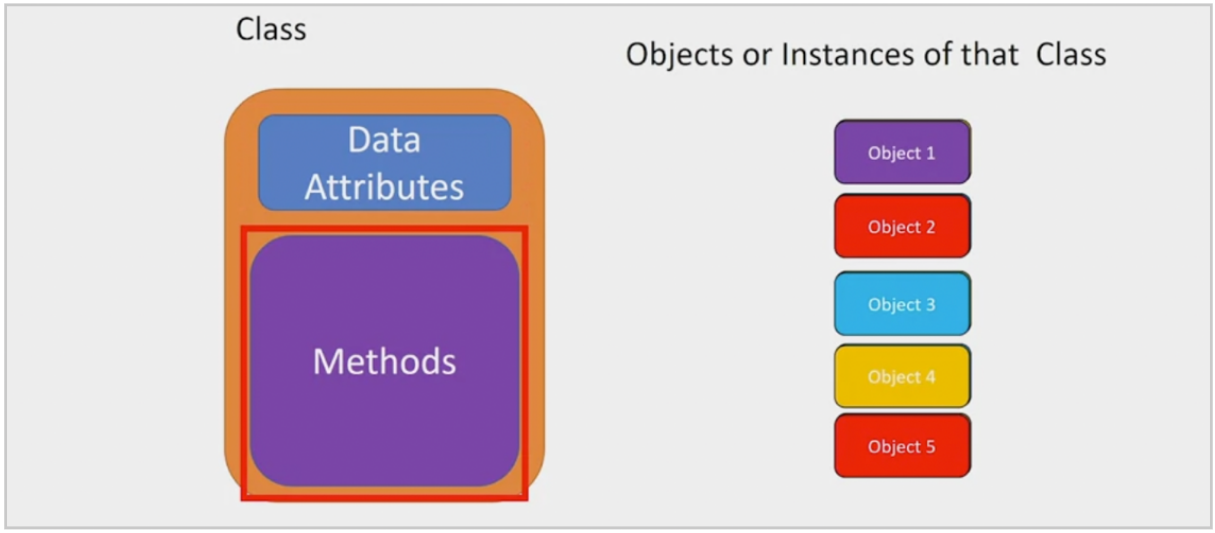


Typing the object's name followed by a dot and the data attribute name gives us the data attribute value, for example, radius. In this case, the radius is 10. We can do the same for color. We can see the relationship between the self parameter and the object.

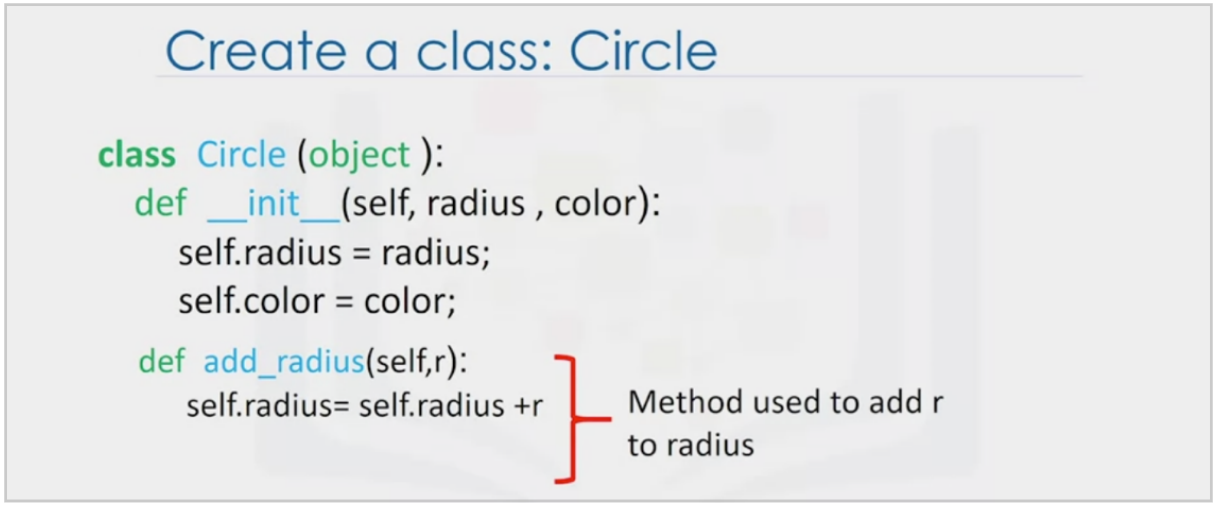


In Python, we can also set or change the data attribute directly. Typing the object's name followed by a dot and the data attribute name, and set it equal to the corresponding value. We can verify that the color data attribute has changed.

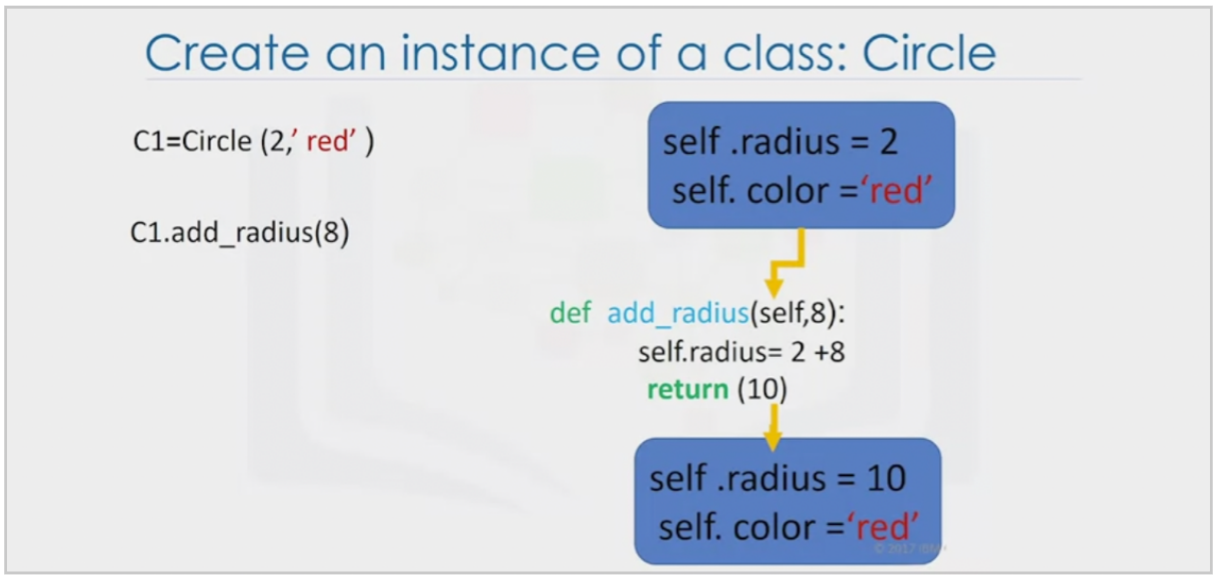
Usually, in order to change the data in an object, we define **methods** in the class. Let's discuss methods. We have seen how data attributes consist of the data defining the objects. **Methods are functions that interact and change the data attributes, changing or using the data attributes of the object**.



Let's say we would like to change the size of a circle. This involves changing the radius attribute. We add a method, add radius to the class circle. The **method has a function that requires the self as well as other parameters**. In this case, we are going to add a value to the radius, we denote that value as r. We are going to add r to the data attribute radius.



Let's see how this part of the code works when we create an object and call the `add_radius` method. As before, we create an object with the object constructor. We pass two arguments to the constructor. The **radius** is set to **2** and the **color** is set to **red**. In the constructor's body, the data attributes are set. We can use the box analogy to see the current state of the object. We call the method by adding a dot followed by the method name, and parentheses. In this case, the argument of the function is the amount we would like to add.



We do not need to worry about the self parameter when calling the method. Just like with the constructor, Python will take care of that for us. In many cases, there may not be any parameters other than self specified in the method's definition. So we don't pass any arguments when calling the function. Internally, the method is called with a value of eight, and the proper self object. The method assigns a new value to self radius. This changes the object, in particular, the radius data attribute. When we call the add_radius method, this changes the object by changing the value of the radius data attribute.

We can add default values to the parameters of a class as constructor. In the labs, we also create the method called drawCircle. See the lab for the implementation of drawCircle. In

the labs, we can create a new object of type circle using the constructor. The color will be red and the radius will be three. We can access the data attribute radius. We can access the attribute color. Finally, we can use the method drawCircle to draw the circle.
Similarly, we can create a new object of type circle. We can access the data attribute of radius. We can access the data attribute color. We can use the method drawCircle to draw the circle. In summary, we have created an object of class circle called RedCircle with a radius attribute of three, and a color attribute of red. We also created an object of class circle called BlueCircle, with a radius attribute of 10 and a color attribute of blue.
In the lab, we have a similar class for rectangle. We can create a new object of type rectangle using the constructor. We can access a data attribute of height. We can also access the data attribute of width. We could do the same for the data attribute of color. We can use the method drawRectangle to draw the rectangle. So we have a class, an object that is a realization or instantiation of that class. For example, we can create two objects of class Circle, or two objects of class Rectangle. The dir function is useful for obtaining the list of data attributes and methods associated with a class. The object you're interested in is passed as an argument. The return value is a list of the objects data attributes. The attribute surrounded by underscores are for internal use, and you shouldn't have to worry about them. The regular looking attributes are the ones you should concern yourself with. These are the objects, methods, and data attributes.

Object and Classes Lab

Objectives

After completing this lab you will be able to:

- Work with classes and objects
- Identify and define attributes and methods

Table of Contents

Introduction to Classes and Objects

Creating a Class

The first step in creating a class is giving it a name. In this notebook, we will create two classes: Circle and Rectangle. We need to determine all the data that make up that class, which we call *attributes*. Think about this step as creating a blue print that we will use to create objects. In figure 1 we see two classes, Circle and Rectangle. Each has their attributes, which are variables. The class Circle has the attribute radius and color, while the Rectangle class has the attribute height and width. Let's use the visual examples of these shapes before we get to the code, as this will help you get accustomed to the vocabulary.

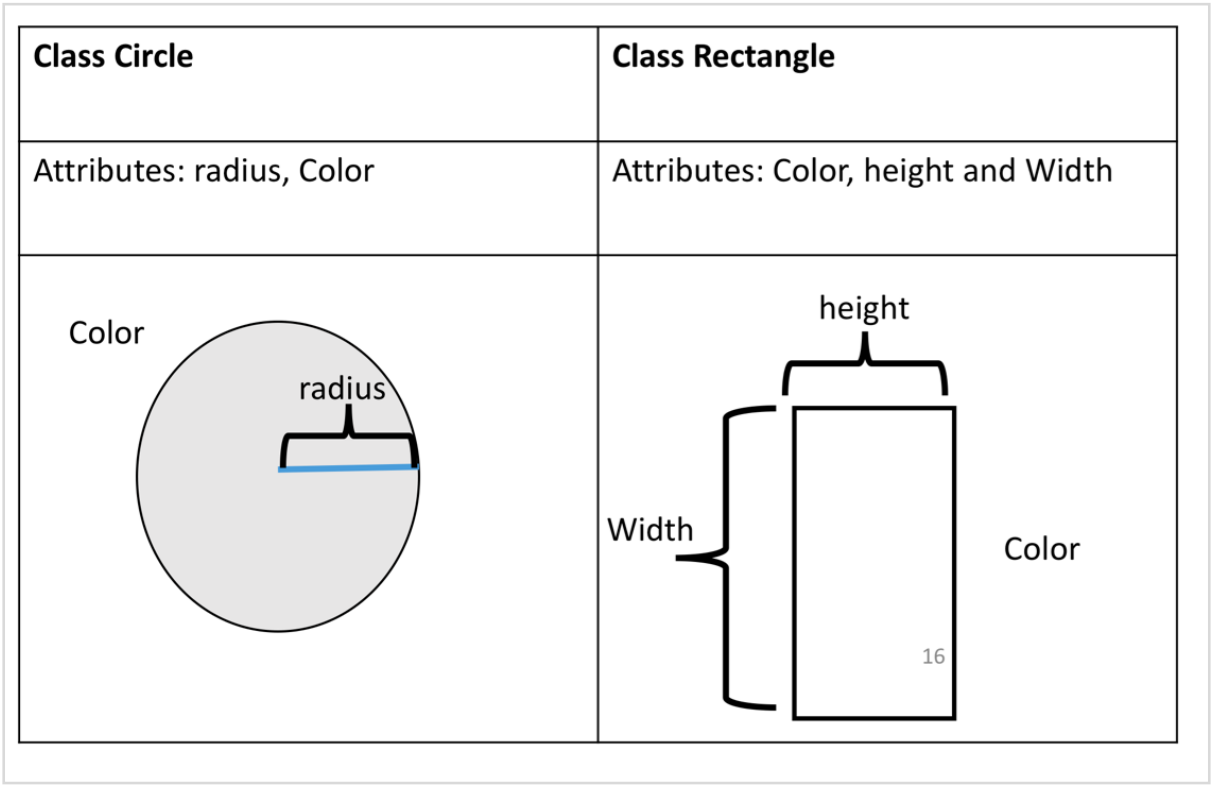


Figure 1: Classes circle and rectangle, and each has their own attributes. The class Circle has the attribute radius and colour, the class Rectangle has the attributes height and width.

Instances of a Class: Objects and Attributes

An instance of an object is the realisation of a class, and in Figure 2 we see three instances of the class circle. We give each object a name: red circle, yellow circle, and green circle. Each object has different attributes, so let's focus on the color attribute for each object.

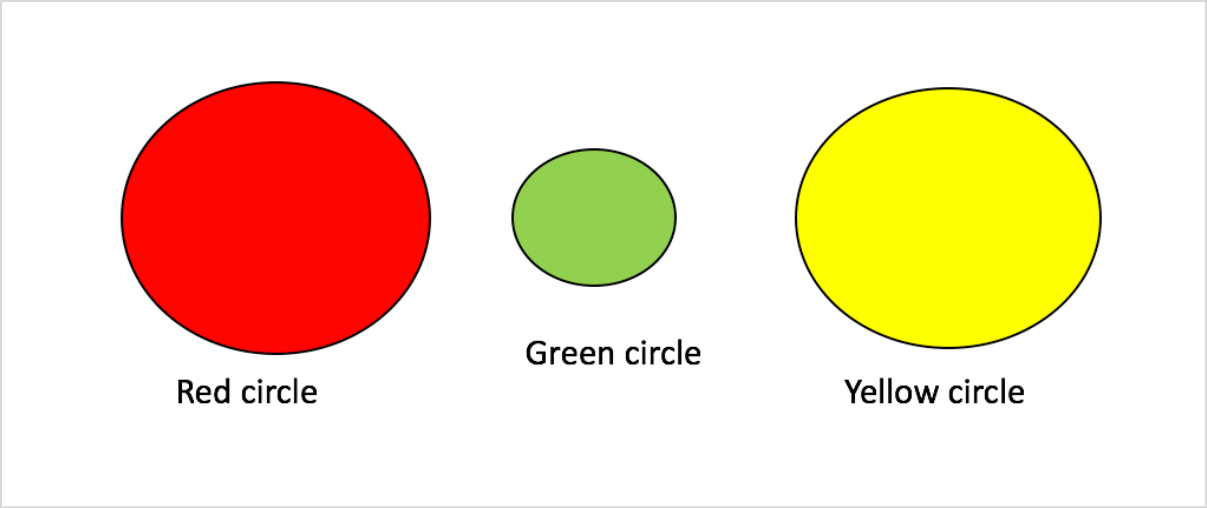


Figure 2: Three instances of the class Circle, or three objects of type Circle.

The colour attribute for the red Circle is the colour red, for the green Circle object the colour attribute is green, and for the yellow Circle the colour attribute is yellow.

Methods

Methods give you a way to change or interact with the object; they are functions that interact with objects. For example, let's say we would like to increase the radius of a circle by a specified amount. We can create a method called `add_radius(r)` that increases the radius by `r`. This is shown in figure 3, where after applying the method to the "orange circle object", the radius of the object increases accordingly. The "dot" notation means to apply the method to the object, which is essentially applying a function to the information in the object.

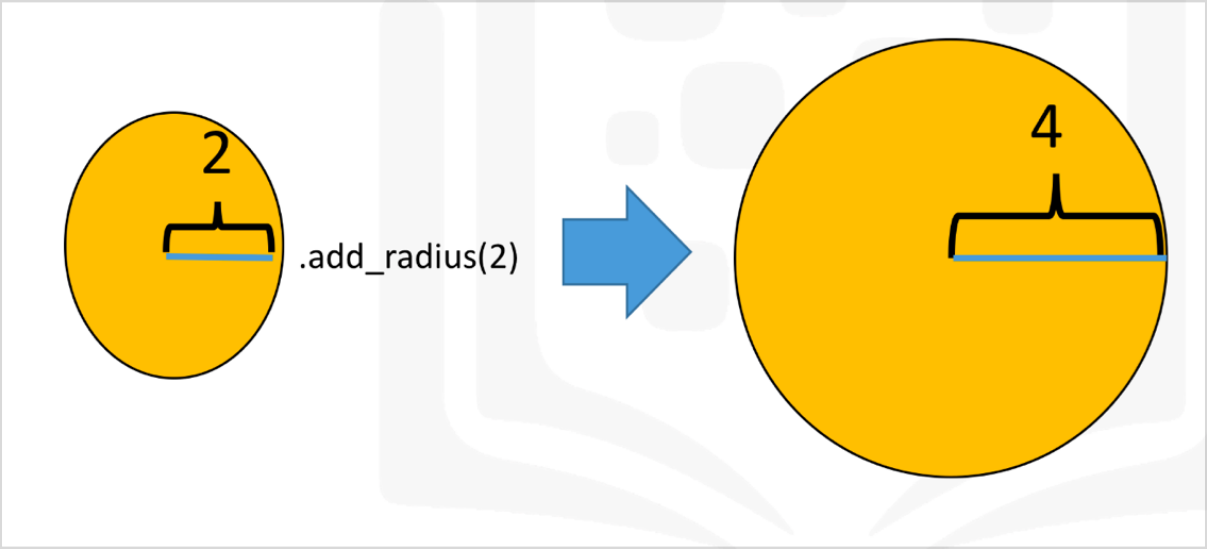


Figure 3: Applying the method `add_radius` to the object orange circle object.

Creating a Class

Now we are going to create a class Circle, but first, we are going to import a library to draw the objects:

```
# Import the library

import matplotlib.pyplot as plt
%matplotlib inline
```

The first step in creating your own class is to use the `class` keyword, then the name of the class as shown in Figure 4. In this course the class parent will always be object:

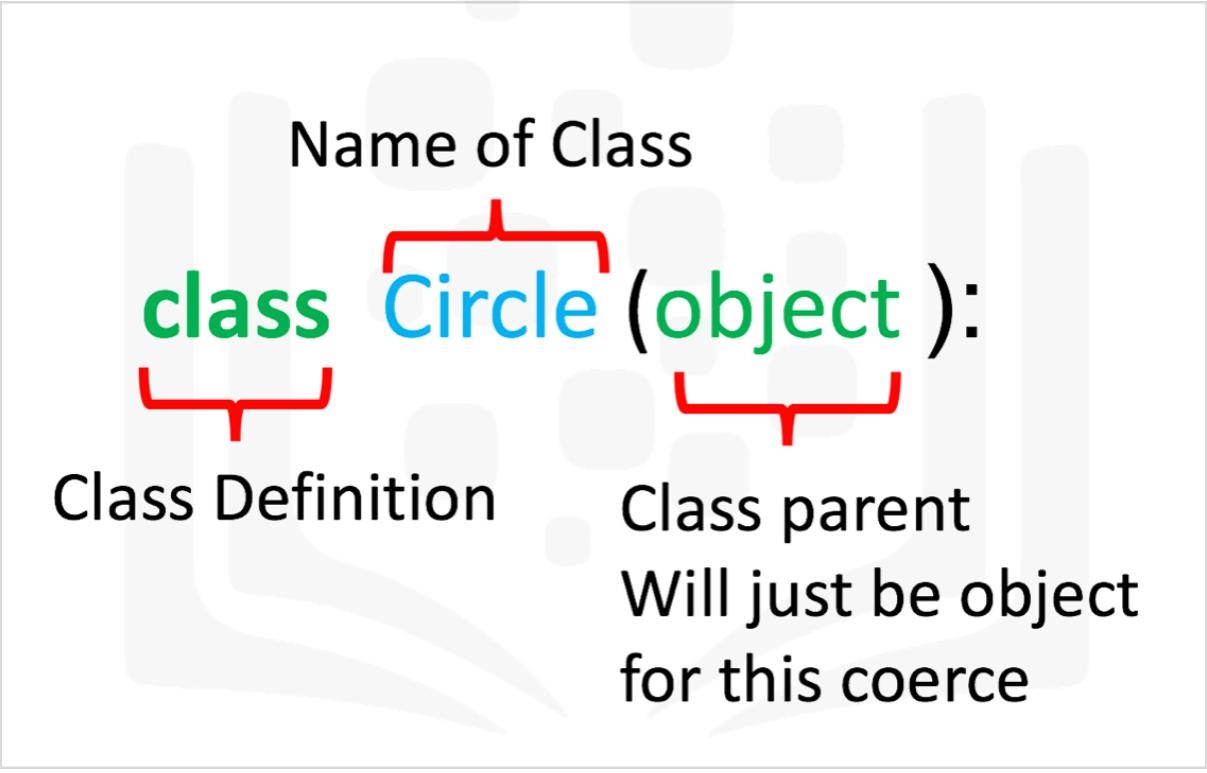


Figure 4: Creating a class Circle.

The next step is a special method called a constructor `__init__`, which is used to initialize the object. The inputs are data attributes. The term `self` contains all the attributes in the set. For example the `self.color` gives the value of the attribute color and `self.radius` will give you the radius of the object. We also have the method `add_radius()` with the parameter `r`, the method adds the value of `r` to the attribute radius. To access the radius we use the syntax `self.radius`. The labeled syntax is summarized in Figure 5:

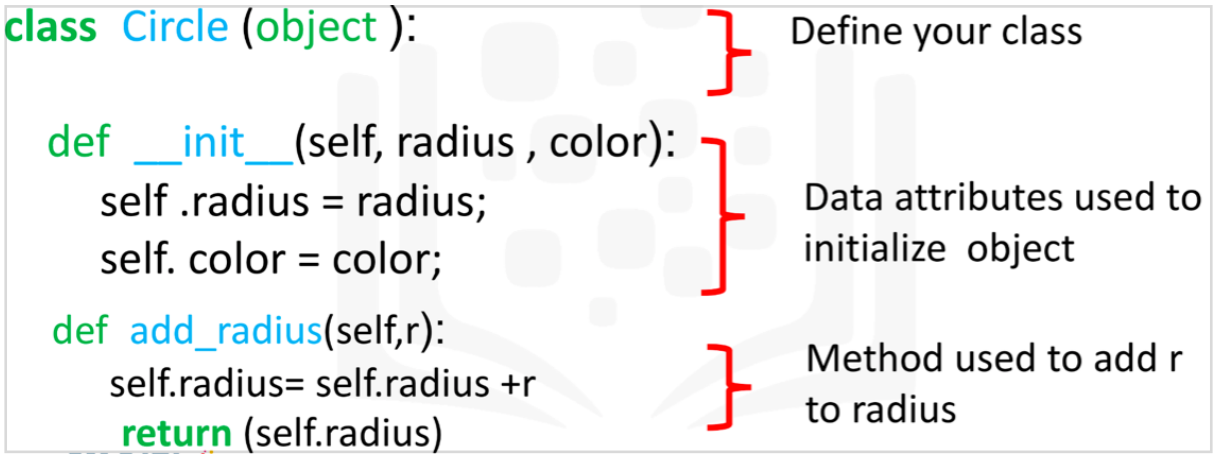


Figure 5: Labeled syntax of the object circle.

The actual object is shown below. We include the method `drawCircle` to display the image of a circle. We set the default radius to 3 and the default colour to blue:

```
# Create a class Circle
class Circle(object):

    # Constructor
    def __init__(self, radius=3, color='blue'):
        self.radius = radius
        self.color = color

    # Method
    def add_radius(self, r):
        self.radius = self.radius + r
        return(self.radius)

    # Method
    def drawCircle(self):
        plt.gca().add_patch(plt.Circle((0, 0), radius=self.radius,
fc=self.color))
        plt.axis('scaled')
        plt.show()
```

Creating an instance of a class Circle

Let's create the object `RedCircle` of type Circle to do the following:

```
# Create an object RedCircle
RedCircle = Circle(10, 'red')
```

We can use the `dir` command to get a list of the object's methods. Many of them are default Python methods.

```
# Find out the methods can be used on the object RedCircle

dir(RedCircle)
```

```
    ['__class__',
     '__delattr__',
     '__dict__',
     '__dir__',
     '__doc__',
     '__eq__',
     '__format__',
     '__ge__',
     '__getattribute__',
     '__gt__',
     '__hash__',
     '__init__',
     '__init_subclass__',
     '__le__',
     '__lt__',
     '__module__',
     '__ne__',
     '__new__',
     '__reduce__',
     '__reduce_ex__',
     '__repr__',
     '__setattr__',
     '__sizeof__',
```

```
    '__str__',
    '__subclasshook__',
    '__weakref__',
    'add_radius',
    'color',
    'drawCircle',
    'radius']
```

We can look at the data attributes of the object:

```
# Print the object attribute radius
RedCircle.radius
```

10

```
# Print the object attribute color
RedCircle.color
```

'red'

We can change the object's data attributes:

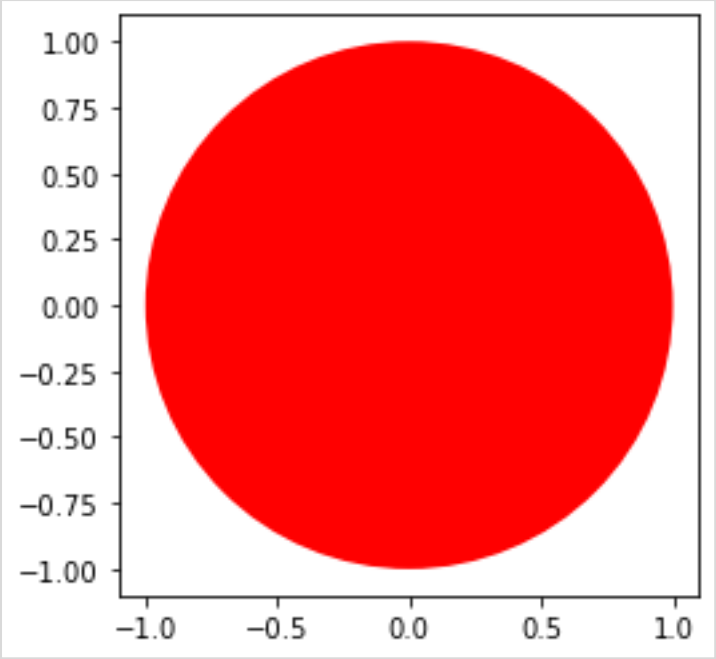
```
# Set the object attribute radius
RedCircle.radius = 1
RedCircle.radius
```

1

We can draw the object by using the method `drawCircle()`:

```
# Call the method drawCircle
```

```
RedCircle.drawCircle()
```



We can increase the radius of the circle by applying the method `add_radius()`. Let's increases the radius by 2 and then by 5:

```
# Use method to change the object attribute radius

print('Radius of object:',RedCircle.radius)
RedCircle.add_radius(2)
print('Radius of object of after applying the method
add_radius(2):',RedCircle.radius)
RedCircle.add_radius(5)
print('Radius of object of after applying the method
add_radius(5):',RedCircle.radius)
```

```
    Radius of object: 1
    Radius of object of after applying the method add_radius(2): 3
    Radius of object of after applying the method add_radius(5): 8
```

Let's create a blue circle. As the default colour is blue, all we have to do is specify what the

radius is:

```
# Create a blue circle with a given radius
BlueCircle = Circle(radius=100)
```

As before, we can access the attributes of the instance of the class by using the dot notation:

```
# Print the object attribute radius
BlueCircle.radius
```

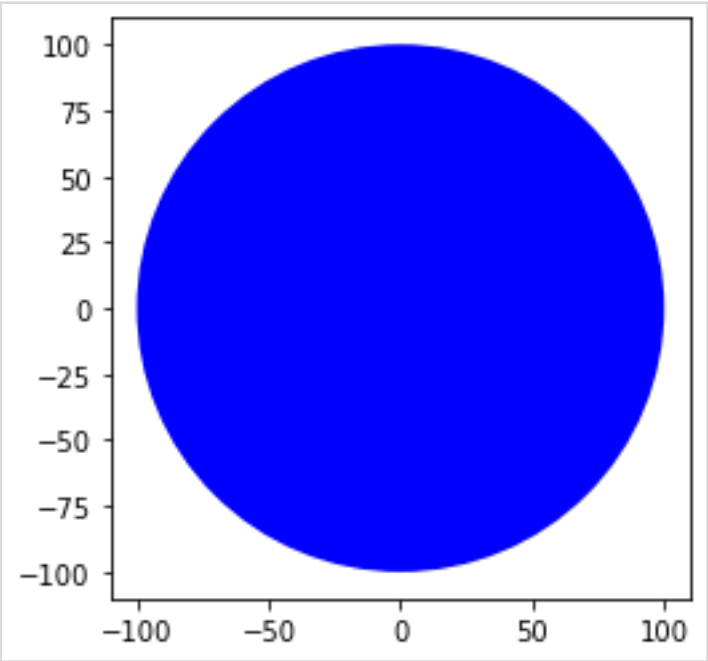
100

```
# Print the object attribute color
BlueCircle.color
```

'blue'

We can draw the object by using the method `drawCircle()`:

```
# Call the method drawCircle
BlueCircle.drawCircle()
```



Compare the x and y axis of the figure to the figure for `RedCircle`; they are different.

The Rectangle Class

Let's create a class rectangle with the attributes of height, width, and color. We will only add the method to draw the rectangle object:

```
# Create a new Rectangle class for creating a rectangle object

class Rectangle(object):

    # Constructor
    def __init__(self, width=2, height=3, color='r'):
        self.height = height
        self.width = width
        self.color = color

    # Method
    def drawRectangle(self):
        plt.gca().add_patch(plt.Rectangle((0, 0), self.width,
self.height ,fc=self.color))
        plt.axis('scaled')
        plt.show()
```

Let's create the object `SkinnyBlueRectangle` of type Rectangle. Its width will be 2 and height will be 3, and the color will be blue:

```
# Create a new object rectangle
SkinnyBlueRectangle = Rectangle(2, 10, 'blue')
```

As before we can access the attributes of the instance of the class by using the dot notation:

```
# Print the object attribute height
SkinnyBlueRectangle.height
```

10

```
# Print the object attribute width
SkinnyBlueRectangle.width
```

2

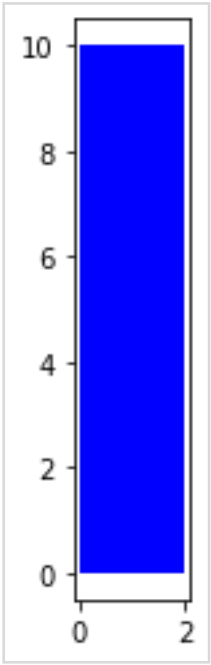
```
# Print the object attribute color
SkinnyBlueRectangle.color
```

'blue'

We can draw the object:

```
# Use the drawRectangle method to draw the shape
```

```
SkinnyBlueRectangle.drawRectangle()
```



Let's create the object `FatYellowRectangle` of type Rectangle:

```
# Create a new object rectangle
FatYellowRectangle = Rectangle(20, 5, 'yellow')
```

We can access the attributes of the instance of the class by using the dot notation:

```
# Print the object attribute height
FatYellowRectangle.height
```

5

```
# Print the object attribute width
FatYellowRectangle.width
```

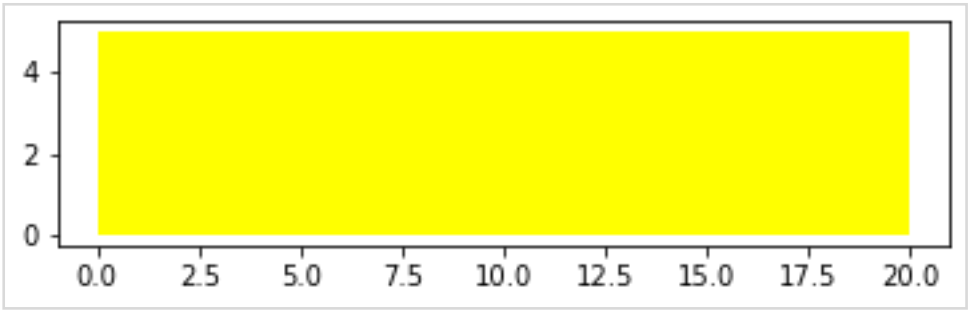
20

```
# Print the object attribute color
FatYellowRectangle.color
```

'yellow'

We can draw the object:

```
# Use the drawRectangle method to draw the shape

FatYellowRectangle.drawRectangle()
```



Exercises

Text Analysis

You have been recruited by your friend, a linguistics enthusiast, to create a utility tool that can perform analysis on a given piece of text. Complete the class 'analysedText' with the following methods -

- Constructor - Takes argument 'text',makes it lower case and removes all punctuation. Assume only the following punctuation is used - period (.), exclamation mark (!), comma (,) and question mark (?). Store the argument in `fmtText`
- `freqAll` - returns a dictionary of all unique words in the text along with the number of their occurences.
- `freqOf` - returns the frequency of the word passed in argument

The skeleton code has been given to you. Docstrings can be ignored for the purpose of the exercise.
Hint: Some useful functions are `replace()`, `lower()`, `split()`, `count()`

```
class analysedText(object):

    def __init__ (self, text):
        # Remove all puntuation
        formattedText =
text.replace('.','').replace('!','').replace(',','').replace('?','')
        # Reduce size
        formattedText = formattedText.lower()

        self.fmtText = formattedText

    def freqAll(self):
        wordList = self.fmtText.split(' ') # trasforma la stringa fornite ed
elaborata nel primo passaggio in lista

        # conteggio quante volte compaionon le stesse parole
        # 1) Creo un dizionario vuoto
        freqMap = {}
        # 2) Creao un ciclo for che mi va a contare le ripetizioni delle parole
inserendole in un dizionario
        for word in set(wordList): # trasformo la lista wordList in un set
            freqMap[word] = wordList.count(word)

        return freqMap # Restituisco il valore del dizionario

    def freqOf(self,word):
        freqDict = self.freqAll()
        if word in freqDict:
            return freqDict[word]
        else:
            return 0
```

```
dir(analysedText)
```

```
    ['__class__',
     '__delattr__',
     '__dict__',
     '__dir__',
     '__doc__',
     '__eq__',
     '__format__',
     '__ge__',
     '__getattribute__',
     '__gt__',
     '__hash__',
     '__init__',
     '__init_subclass__',
     '__le__',
     '__lt__',
     '__module__',
     '__ne__',
     '__new__',
     '__reduce__',
     '__reduce_ex__',
     '__repr__',
     '__setattr__',
     '__sizeof__',
     '__str__',
     '__subclasshook__',
     '__weakref__',
     'freqAll',
     'freqOf']
```

Execute the block below to check your progress.

```
import sys
```

```
sampleMap = {'eirmod': 1,'sed': 1, 'amet': 2, 'diam': 5, 'consetetur': 1,
'labore': 1, 'tempor': 1, 'dolor': 1, 'magna': 2, 'et': 3, 'nonumy': 1,
'ipsum': 1, 'lorem': 2}

def testMsg(passed):
    if passed:
       return 'Test Passed'
    else :
       return 'Test Failed'

print("Constructor: ")
try:
    samplePassage = analysedText("Lorem ipsum dolor! diam amet, consetetur
Lorem magna. sed diam nonumy eirmod tempor. diam et labore? et diam magna. et
diam amet.")
    print(testMsg(samplePassage.fmtText == "lorem ipsum dolor diam amet
consetetur lorem magna sed diam nonumy eirmod tempor diam et labore et diam
magna et diam amet"))
except:
    print("Error detected. Recheck your function " )
print("freqAll: ")
try:
    wordMap = samplePassage.freqAll()
    print(testMsg(wordMap==sampleMap))
except:
    print("Error detected. Recheck your function " )
print("freqOf: ")
try:
    passed = True
    for word in sampleMap:
        if samplePassage.freqOf(word) != sampleMap[word]:
            passed = False
            break
    print(testMsg(passed))

except:
    print("Error detected. Recheck your function  " )
```

Constructor:
Test Passed
freqAll:
Test Passed
freqOf:
Test Passed