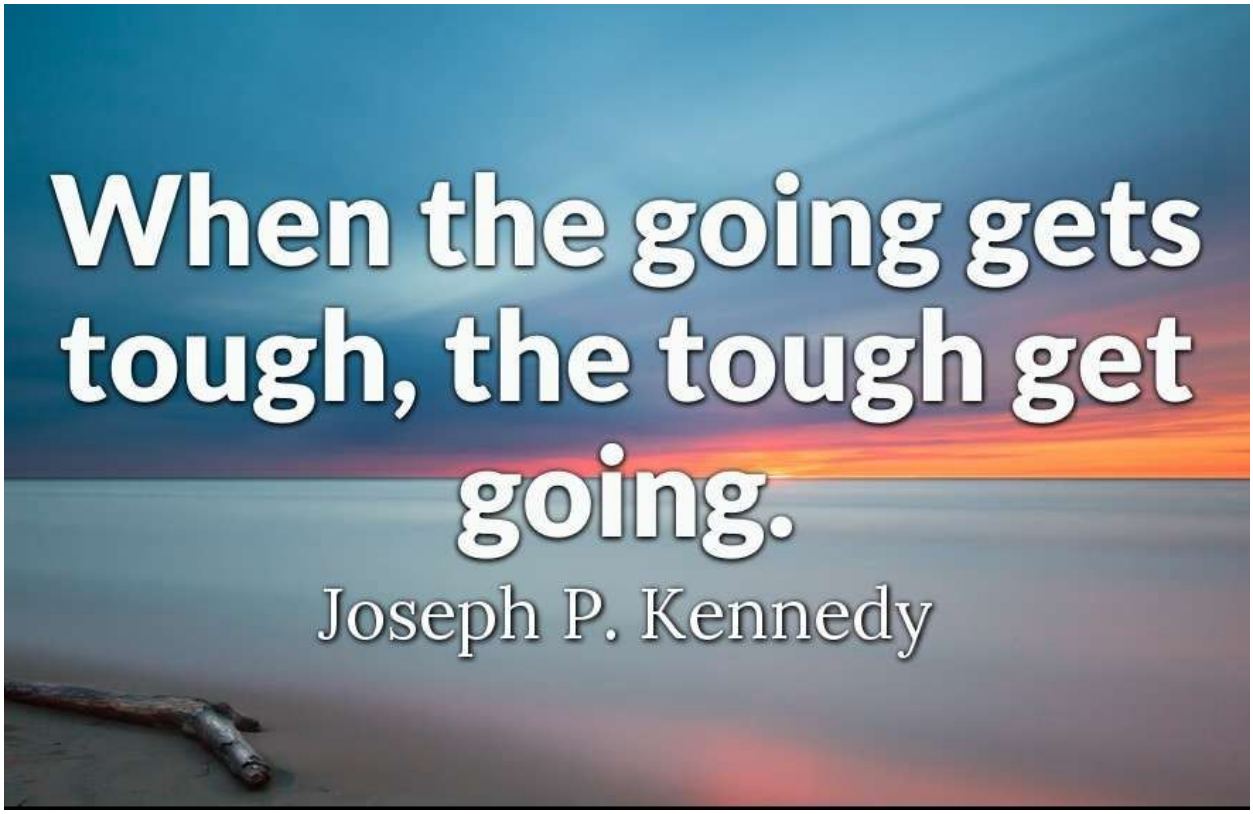
LEARNING PYTHON

-Diwash Sapkota



BASIC

Introduction

Python is a high-level programming language, with applications in numerous areas, including web programming, scripting, scientific computing, and artificial intelligence.

It is very popular and used by organizations such as Google, NASA, the CIA, and Disney.

Python is processed at runtime by the interpreter. There is no need to compile the program before executing it.

The three major versions of Python are 1.x, 2.x and 3.x. These are subdivided into minor versions, such as 2.7 and 3.3. Code written for Python 3.x is guaranteed to work in all future versions. Both Python Version 2.x and 3.x are used currently.

This course covers Python 3.x, but it isn't hard to change from one version to another.

Python has several different implementations, written in various languages.

The version used in this course, CPython, is the most popular by far.

An interpreter is a program that runs scripts written in an interpreted language such as Python.

Simple Operations:

Python has the capability of carrying out calculations.

Enter a calculation directly into the Python console, and it will output the answer.

```
>>> 2 + 2
4
>>> 5 + 4 - 3
6
```

The spaces around the plus and minus signs here are optional (the code would work without them), but they make it easier to read.

Python also carries out multiplication and division, using an asterisk to indicate multiplication and a forward slash to indicate division.

Use parentheses to determine which operations are performed first.

```
>>> 2 * (3 + 4)
14
>>> 10 / 2
5.0
```

Using a single slash to divide numbers produces a decimal (or float, as it is called in programming). We'll have more about floats in a later lesson.

The minus sign indicates a negative number.

Operations are performed on negative numbers, just as they are on positive ones.

```
>>> -7
-7
>>> (-7 + 2) * (-4)
20
```

The plus signs can also be put in front of numbers, but this has no effect, and is mostly used to emphasize that a number is positive to increase readability of code.

Floats:

Floats are used in Python to represent numbers that aren't integers.

Some examples of numbers that are represented as floats are 0.5 and -7.8237591.

They can be created directly by entering a number with a decimal point, or by using operations such as division on integers. Extra zeros at the number's end are ignored.

```
>>> 3/4
0.75
>>> 9.8765000
9.8765
```

Computers can't store floats perfectly accurately, in the same way that we can't write down the complete decimal expansion of 1/3 (0.3333333333333333...). Keep this in mind, because it often leads to infuriating bugs!

As you saw previously, dividing any two integers produces a float.

A float is also produced by running an operation on two floats, or on a float and an integer.

```
>>> 8 / 2
4.0
>>> 6 * 7.0
42.0
>>> 4 + 1.65
```

5.65

A float can be added to an integer, because Python silently converts the integer to a float. However, this implicit conversion is the exception rather than the rule in Python - usually you have to convert values manually if you want to operate on them.

OTHER NUMERICAL OPERATORS:

Exponentiation

Besides addition, subtraction, multiplication, and division, Python also supports exponentiation, which is the raising of one number to the power of another. This operation is performed using two asterisks.

*>>> 2**5*

32

*>>> 9 ** (1/2)*

3.0

You can chain exponentiations together. In other words, you can rise a number to multiple powers. For example, 2 3 2.

Quotient & Remainder

To determine the quotient and remainder of a division, use the floor division and modulo operators, respectively.

Floor division is done using two forward slashes.

The modulo operator is carried out with a percent symbol (%).

These operators can be used with both floats and integers.

This code shows that 6 goes into 20 three times, and the remainder when 1.25 is divided by 0.5 is 0.25.

>>> 20 // 6

3

>>> 1.25 % 0.5

0.25

In the example above, 20 % 6 will return 2, because 3*6+2 is equal to 20.

Strings

If you want to use text in Python, you have to use a string.

A string is created by entering text between two single or double quotation marks.
When the Python console displays a string, it generally uses single quotes. The delimiter used for a string doesn't affect how it behaves in any way.
>>> "Python is fun!"
'Python is fun!'
>>> 'Always look on the bright side of life'
'Always look on the bright side of life'

There is another string type in Python called docstrings that is used for block commenting, but it is actually a string. You will learn about this in future lessons.

Some characters can't be directly included in a string. For instance, double quotes can't be directly included in a double quote string; this would cause it to end prematurely.

Characters like these must be escaped by placing a backslash before them. Other common characters that must be escaped are newlines and backslashes.

Double quotes only need to be escaped in double quote strings, and the same is true for single quote strings.
>>> 'Brian\'s mother: He\'s not the Messiah. He\'s a very naughty boy!'
'Brian's mother: He's not the Messiah. He's a very naughty boy!'

\n represents a new line.

Backslashes can also be used to escape tabs, arbitrary Unicode characters, and various other things that can't be reliably printed. These characters are known as escape characters.

Newlines

Python provides an easy way to avoid manually writing "\n" to escape newlines in a string. Create a string with three sets of quotes, and newlines that are created by pressing Enter are automatically escaped for you.

>>> """Customer: Good morning.

Owner: Good morning, Sir. Welcome to the National Cheese Emporium."""

'Customer: Good morning.\nOwner: Good morning, Sir. Welcome to the National Cheese Emporium.'

As you can see, the \n was automatically put in the output, where we pressed Enter.

Simple Input & Output

Output

Usually, programs take input and process it to produce output. In Python, you can use the print function to produce output. This displays a textual representation of something to the screen.

>>> print(1 + 1)

2

>>> print("Hello\nWorld!")

Hello

World!

Input

To get input from the user in Python, you can use the intuitively named input function.
The function prompts the user for input, and returns what they enter as a string (with the contents automatically escaped).

>>> input("Enter something please: ")

Enter something please: This is what\nthe user enters!

'This is what\\nthe user enters!'

The print and input functions aren't very useful at the Python console, which automatically does input and output. However, they are very useful in actual programs.

Concatenation

As with integers and floats, strings in Python can be added, using a process called concatenation, which can be done on any two strings.

When concatenating strings, it doesn't matter whether they've been created with single or double quotes.

```
>>> "Spam" + 'eggs'
'Spameggs'
>>> print("First string" + ", " + "second string")
First string, second string
```

You can’t concatenate strings with numbers (integers). Find out why in the next lesson.

Even if your strings contain numbers, they are still added as strings rather than integers. Adding a string to a number produces an error, as even though they might look similar, they are two different entities.

```
>>> "2" + "2"
'22'
>>> 1 + '2' + 3 + '4'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

String Operations

Strings can also be multiplied by integers. This produces a repeated version of the original string. The order of the string and the integer doesn't matter, but the string usually comes first.

Strings can't be multiplied by other strings. Strings also can't be multiplied by floats, even if the floats are whole numbers.

```
>>> print("spam" * 3)
spamspamspam
>>> 4 * '2'
'2222'
>>> '17' * '87'
TypeError: can't multiply sequence by non-int of type 'str
```

*>>> 'pythonisfun' * 7.0*

TypeError: can't multiply sequence by non-int of type 'float'

Try to multiply a string by 0 (zero) and see what happens.

Type Conversion

In Python, it's impossible to complete certain operations due to the types involved. For instance, you can't add two strings containing the numbers 2 and 3 together to produce the integer 5, as the operation will be performed on strings, making the result '23'.

The solution to this is type conversion.

In that example, you would use the int function.

>>> "2" + "3"

'23'

>>> int("2") + int("3")

5

In Python, the types we have used so far have been integers, floats, and strings. The functions used to convert to these are int, float and str, respectively.

Another example of type conversion is turning user input (which is a string) to numbers (integers or floats), to allow for the performance of calculations.

>>> float(input("Enter a number: ")) + float(input("Enter another number: "))

Enter a number: 40

Enter another number: 2

42.0

Passing non-integer or float values will cause an error.

Variables

Variables play a very important role in most programming languages, and Python is no exception. A variable allows you to store a value by assigning it to a name, which can be used to refer to the value later in the program.

To assign a variable, use one equals sign. Unlike most lines of code we've looked at so far, it doesn't produce any output at the Python console.

>>> x = 7

>>> print(x)

7

>>> print(x + 3)

10

>>> print(x)

7

You can use variables to perform corresponding operations, just as you did with numbers and strings. As you can see, the variable stores its value throughout the program.

Variables can be reassigned as many times as you want, in order to change their value.

In Python, variables don't have specific types, so you can assign a string to a variable, and later assign an integer to the same variable.

>>> x = 123.456

>>> print(x)

123.456

>>> x = "This is a string"

>>> print(x + "!")

This is a string!

However, it is not good practice. To avoid mistakes, try to avoid overwriting the same variable with different data types.

Variable Names

Certain restrictions apply in regard to the characters that may be used in Python variable names. The only characters that are allowed are letters, numbers, and underscores. Also, they can't start with numbers.

Not following these rules results in errors.

>>> this_is_a_normal_name = 7

>>> 123abc = 7

SyntaxError: invalid syntax

>>> spaces are not allowed

SyntaxError: invalid syntax

Python is a case sensitive programming language. Thus, Lastname and lastname are two different variable names in Python.

Trying to reference a variable you haven't assigned to causes an error.

You can use the del statement to remove a variable, which means the reference from the name to the value is deleted, and trying to use the variable causes an error. Deleted variables can be reassigned to later as normal.

```
>>> foo = "a string"
>>> foo
'a string'
>>> bar
NameError: name 'bar' is not defined

>>> del foo
>>> foo
NameError: name 'foo' is not defined
```

You can also take the value of the variable from the user input.

```
>>> foo = input("Enter a number: ")
Enter a number: 7
>>> print(foo)
7
```

In-Place Operators

In-place operators allow you to write code like 'x = x + 3' more concisely, as 'x += 3'.

The same thing is possible with other operators such as -, *, / and % as well.

```
>>> x = 2
>>> print(x)
2
>>> x += 3
>>> print(x)
5
```

These operators can be used on types other than numbers, as well, such as strings.

```
>>> x = "spam"
>>> print(x)
spam

>>> x += "eggs"
>>> print(x)
```

spameggs

Many other languages have special operators such as '++' as a shortcut for 'x += 1'. Python does not have these.

Using an Editor

So far, we've only used Python with the console, entering and running one line of code at a time.

Actual programs are created differently; many lines of code are written in a file, and then executed with the Python interpreter.

In IDLE, this can be done by creating a new file, entering some code, saving the file, and running it. This can be done either with the menus or with the keyboard shortcuts Ctrl-N, Ctrl-S and F5.

Each line of code in the file is interpreted as though you entered it one line at a time at the console.

```
x = 7
x = x + 2
print(x)
```

CONTROL STRUCTURES

Booleans

Another type in Python is the Boolean type. There are two Boolean values: True and False. They can be created by comparing values, for instance by using the equal operator ==.

>>> my_boolean = True

>>> my_boolean

True

>>> 2 == 3

False

>>> "hello" == "hello"

True

Be careful not to confuse assignment (one equals sign) with comparison (two equals signs).

Comparison

Another comparison operator, the not equal operator (!=), evaluates to True if the items being compared aren't equal, and False if they are.

>>> 1 != 1

False

>>> "eleven" != "seven"

True

>>> 2 != 10

True

Python also has operators that determine whether one number (float or integer) is greater than or smaller than another. These operators are > and < respectively.

>>> 7 > 5

True

>>> 10 < 10

False

Greater than and smaller than operators can also be used to compare strings lexicographically (the alphabetical order of words is based on the alphabetical order of their component letters).

if Statements

You can use if statements to run code if a certain condition holds.

If an expression evaluates to True, some statements are carried out. Otherwise, they aren't carried out.

An if statement looks like this:

```
if expression:
statements
```

Python uses indentation (white space at the beginning of a line) to delimit blocks of code. Other languages, such as C, use curly braces to accomplish this, but in Python indentation is mandatory; programs won't work without it. As you can see, the statements in the if should be indented. Here is an example if statement:

```
if 10 > 5:
    print("10 greater than 5")
print("Program ended")
```

To perform more complex checks, if statements can be nested, one inside the other.

This means that the inner if statement is the statement part of the outer one. This is one way to see whether multiple conditions are satisfied. For example:

```
num = 12
if num > 5:
    print("Bigger than 5")
    if num <=47:
        print("Between 5 and 47")
```

else Statements

An else statement follows an if statement, and contains code that is called when the if statement evaluates to False.

As with if statements, the code inside the block should be indented.

```
x = 4
if x == 5:
    print("Yes")
else:
    print("No")
```

elif Statements

The elif (short for else if) statement is a shortcut to use when chaining if and else statements. A series of if elif statements can have a final else block, which is called if none of the if or elif expressions is True.

For example:

```
num = 7
if num == 5:
    print("Number is 5")
elif num == 11:
    print("Number is 11")
elif num == 7:
    print("Number is 7")
else:
    print("Number isn't 5, 11 or 7")
```

Boolean Logic

Boolean logic is used to make more complicated conditions for if statements that rely on more than one condition.

Python's Boolean operators are and, or, and not.

The and operator takes two arguments, and evaluates as True if, and only if, both of its arguments are True. Otherwise, it evaluates to False.

```
>>> 1 == 1 and 2 == 2
True
>>> 1 == 1 and 2 == 3
False
>>> 1 != 1 and 2 == 2
False
>>> 2 < 1 and 3 >  6
False
```

Python uses words for its Boolean operators, whereas most other languages use symbols such as &&, || and !.

Boolean Or

The or operator also takes two arguments. It evaluates to True if either (or both) of its arguments are True, and False if both arguments are False.

>>> 1 == 1 or 2 == 2

True

>>> 1 == 1 or 2 == 3

True

>>> 1 != 1 or 2 == 2

True

>>> 2 < 1 or 3 > 6

False

Boolean Not

Unlike other operators we've seen so far, not only takes one argument, and inverts it.
The result of not True is False, and not False goes to True.

>>> not 1 == 1

False

>>> not 1 > 7

True

You can chain multiple conditional statements in an if statement using the Boolean operators.

Operator Precedence

Operator precedence is a very important concept in programming. It is an extension of the mathematical idea of order of operations (multiplication being performed before addition, etc.) to include other operators, such as those in Boolean logic. The below code shows that == has a higher precedence than or:

>>> False == False or True

True

>>> False == (False or True)

False

>>> (False == False) or True

True

Python's order of operations is the same as that of normal mathematics: parentheses first, then exponentiation, then multiplication/division, and then addition/subtraction.

Operator Precedence

The following table lists all of Python's operators, from highest precedence to lowest.

Operator	Description
**	Exponentiation (raise to the power)
~, +, -	Complement, unary plus and minus (method names for the last two are +@ and -@)
*, /, %, //	Multiply, divide, modulo and floor division
+, -	Addition and subtraction
>>, <<	Right and left bitwise shift
&	Bitwise 'AND'
^	Bitwise exclusive 'OR'
|	Bitwise 'OR'
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparison operators, equality operators, membership and identity operators
not	Boolean 'NOT'
and	Boolean 'AND'
or	Boolean 'OR'
=, %=, /=, //=, -=, +=, *=, **=	Assignment operators

Operators in the same box have the same precedence.

while Loops

An if statement is run once if its condition evaluates to True, and never if it evaluates to False.

A while statement is similar, except that it can be run more than once. The statements inside it are repeatedly executed, as long as the condition holds. Once it evaluates to False, the next section of code is executed.

Below is a while loop containing a variable that counts up from 1 to 5, at which point the loop terminates.

```
i = 1
while i <=5:
    print(i)
    i = i + 1
 print("Finished!")
```

Result:

```
>>>
1
2
3
4
5
Finished!
>>>
```

The code in the body of a **while** loop is executed repeatedly. This is called **iteration**.

while Loops

The infinite loop is a special kind of while loop; it never stops running. Its condition always remains True. An example of an infinite loop:

```
while 1==1:
    print("In the loop")
```

This program would indefinitely print "In the loop".

You can stop the program's execution by using the Ctrl-C shortcut or by closing the program.

break

To end a while loop prematurely, the break statement can be used.

When encountered inside a loop, the break statement causes the loop to finish immediately.

```
i = 0
while 1==1:
    print(i)
    i = i + 1
    if i >= 5:
        print("Breaking")
        break
print("Finished")
Result:>>>
0
1
2
3
4
Breaking
Finished
>>>
```

Using the **break** statement outside of a loop causes an error.

Continue

Another statement that can be used within loops is continue. Unlike break, continue jumps back to the top of the loop, rather than stopping it.

```
i = 0
while True:
    i = i +1
    if i == 2:
        print("Skipping 2")
        continue
    if i == 5:
        print("Breaking")
        break
    print(i)
print("Finished")


Result:>>>
1
Skipping 2
3
4
Breaking
Finished
>>>
```

Basically, the continue statement stops the current iteration and continues with the next one.

Using the continue statement outside of a loop causes an error.

Lists

Lists are another type of object in Python. They are used to store an indexed list of items. A list is created using square brackets with commas separating items. A certain item in the list can be accessed by using its index in square brackets. For example:

words = ["Hello", "world", "!"]
print(words[0])
print(words[1])
print(words[2])

The first list item's index is **0**, rather than 1, as might be expected.

An empty list is created with an empty pair of square brackets.
empty_list = []
print(empty_list)

Most of the time, a comma won't follow the last item in a list. However, it is perfectly valid to place one there, and it is encouraged in some cases.

Typically, a list will contain items of a single item type, but it is also possible to include several different types. Lists can also be nested within other lists.

number = 3
things = ["string", 0, [1, 2, number], 4.56]
print(things[1])
print(things[2])
print(things[2][2])

Result:
>>>
0
[1, 2, 3]
3
>>>

Lists of lists are often used to represent 2D grids, as Python lacks the multidimensional arrays that would be used for this in other languages.

Indexing out of the bounds of possible list values causes an IndexError. Some types, such as strings, can be indexed like lists. Indexing strings behaves as though you are indexing a list containing each character in the string. For other types, such as integers, indexing them isn't possible, and it causes a TypeError.

List Operations

The item at a certain index in a list can be reassigned. For example:

nums = [7, 7, 7, 7, 7]

nums[2] = 5

print(nums)

Lists can be added and multiplied in the same way as strings. For example:

nums = [1, 2, 3]

print(nums + [4, 5, 6])

*print(nums * 3)*

Lists and strings are similar in many ways - strings can be thought of as lists of characters that can't be changed.

To check if an item is in a list, the in operator can be used. It returns True if the item occurs one or more times in the list, and False if it doesn't.

words = ["spam", "egg", "spam", "sausage"]

print("spam" in words)

print("egg" in words)

print("tomato" in words)

The in operator is also used to determine whether or not a string is a substring of another string.

To check if an item is not in a list, you can use the not operator in one of the following ways:

nums = [1, 2, 3]

*print(**not** 4 in nums) #True*

*print(4 **not** in nums) #True*

*print(**not** 3 in nums) #False*

*print(3 **not** in nums) #False*

List Functions

Another way of altering lists is using the append method. This adds an item to the end of an existing list.

```
nums = [1, 2, 3]
nums.append(4)
print(nums) #1 2 3 4
```

The dot before append is there because it is a method of the list class. Methods will be explained in a later lesson.

List Functions

To get the number of items in a list, you can use the len function.

```
nums = [1, 3, 5, 2, 4]
print(len(nums))
```

Unlike append, len is a normal function, rather than a method. This means it is written before the list it is being called on, without a dot.

The insert method is similar to append, except that it allows you to insert a new item at any position in the list, as opposed to just at the end.

```
words = ["Python", "fun"]
index = 1
words.insert(index, "is")
print(words)
```

The index method finds the first occurrence of a list item and returns its index. If the item isn't in the list, it raises a ValueError.

```
letters = ['p', 'q', 'r', 's', 'p', 'u']
print(letters.index('r'))
print(letters.index('p'))
print(letters.index('z'))
```

There are a few more useful functions and methods for lists.

max(list): Returns the list item with the maximum value

min(list): Returns the list item with minimum value

list.**count**(obj): Returns a count of how many times an item occurs in a list

list.**remove**(obj): Removes an object from a list

list.**reverse**(): Reverses objects in a list

Range

The range function creates a sequential list of numbers. The code below generates a list containing all of the integers, up to 10.

```
numbers = list(range(10))
print(numbers)
```

The call to list is necessary because range by itself creates a range object, and this must be converted to a list if you want to use it as one.

If range is called with one argument, it produces an object with values from 0 to that argument. If it is called with two arguments, it produces values from the first to the second. For example:

```
numbers = list(range(3, 8))
print(numbers)
print(range(20) == range(0, 20))
```

range can have a third argument, which determines the interval of the sequence produced. This third argument must be an integer.

```
numbers = list(range(5, 20, 2))
print(numbers)
```

Loops

Sometimes, you need to perform code on each item in a list. This is called iteration, and it can be accomplished with a while loop and a counter variable. For example:

```
words = ["hello", "world", "spam", "eggs"]
counter = 0
max_index = len(words) - 1
while (counter <= max_index):
    word = words[counter]
    print(word + "!")
    counter = counter + 1
```

The example above iterates through all items in the list, accesses them using their indices, and prints them with exclamation marks.

for Loop

Iterating through a list using a while loop requires quite a lot of code, so Python provides the for loop as a shortcut that accomplishes the same thing. The same code from the previous example can be written with a for loop, as follows:

```
words = ["hello", "world", "spam", "eggs"]
for word in words:
    print(word + "!")
```

The for loop in Python is like the foreach loop in other languages.

The for loop is commonly used to repeat some code a certain number of times. This is done by combining for loops with range objects.

```
for i in range(5):
 print("hello!")
```

You don't need to call list on the range object when it is used in a for loop, because it isn't being indexed, so a list isn't required.

Creating a Calculator

This lesson is about an example Python project: a simple calculator.

Each part explains a different section of the program.

The first section is the overall menu. This keeps on accepting user input until the user enters "quit", so a while loop is used.

```
while True:
    print("Options:")
    print("Enter 'add' to add two numbers")
    print("Enter 'subtract' to subtract two numbers")
    print("Enter 'multiply' to multiply two numbers")
    print("Enter 'divide' to divide two numbers")
    print("Enter 'quit' to end the program")
    user_input = input(": ")
    if user_input == "quit":
        break
    elif user_input == "add":
        ...
    elif user_input == "subtract":
        ...
    elif user_input == "multiply":
        ...
    elif user_input == "divide":
    ...
    else:
    print("Unknown input")
```

The code above is the starting point for our program. It accepts user input, and compares it to the options in the if/elif statements. The break statement is used to stop the while loop, in case the user inputs "quit".

```
while(True):
    print("options:")
    print("Enter 'quit' to exit the program")
    print("Enter 'add' to perform addition")
    print("Enter 'sub' to perform addition")
    print("Enter 'multiply' to perform addition")
    print("Enter 'division' to perform addition")
    user_input = input(": ")
    if user_input == "quit":
        break
    elif user_input == "add":
        num1 = float(input("Enter the first number: "))
        num2 = float(input("Enter the second number: "))
        print("The result of addition of two numbers is ",num1+num2)
    elif user_input == "sub":
        num1 = float(input("Enter the first number: "))
        num2 = float(input("Enter the second number: "))
        print("The result of subtraction of two numbers is ",num1-num2)
    elif user_input == "multiply":
        num1 = float(input("Enter the first number: "))
        num2 = float(input("Enter the second number: "))
        print("The result of multiplication of two numbers is ",num1*num2)
    elif user_input == "division":
```

```
    num1 = float(input("Enter the first number: "))
    num2 = float(input("Enter the second number: "))
    print("The result of division of num1 by num2 is ",num1/num2)
else:
    print("Unknown input")
```

EXCEPTION AND FILES

Exceptions

Different exceptions are raised for different reasons.

Common exceptions:

ImportError: an import fails;

IndexError: a list is indexed with an out-of-range number;

NameError: an unknown variable is used;

SyntaxError: the code can't be parsed properly;

TypeError: a function is called on a value of an inappropriate type;

ValueError: a function is called on a value of the correct type, but with an inappropriate value.

Python has several other built-in exceptions, such as ZeroDivisionError and OSError. Third-party libraries also often define their own exceptions.

Exception Handling

To handle exceptions, and to call code when an exception occurs, you can use a try/except statement.

The try block contains code that might throw an exception. If that exception occurs, the code in the try block stops being executed, and the code in the except block is run. If no error occurs, the code in the except block doesn't run. For example:

```
try:
  num1 = 7
  num2 = 0
  print (num1 / num2)
  print("Done calculation")
except ZeroDivisionError:
  print("An error occurred")
  print("due to zero division")
```

In the code above, the except statement defines the type of exception to handle (in our case, the ZeroDivisionError).

A try statement can have multiple different except blocks to handle different exceptions. Multiple exceptions can also be put into a single except block using parentheses, to have the except block handle all of them.

```
try:
  variable = 10
  print(variable + "hello")
  print(variable / 2)
except ZeroDivisionError:
  print("Divided by zero")
except (ValueError, TypeError):
  print("Error occurred")
```

An except statement without any exception specified will catch all errors. These should be used sparingly, as they can catch unexpected errors and hide programming mistakes. For example:

```
try:
  word = "spam"
  print(word / 0)
except:
  print("An error occurred")

RESULT>>
An error occurred
>>
```

Exception handling is particularly useful when dealing with user input.

To ensure some code runs no matter what errors occur, you can use a finally statement. The finally statement is placed at the bottom of a try/except statement. Code within a finally statement always runs after execution of the code in the try, and possibly in the except, blocks.

```
try:
  print("Hello")
  print(1 / 0)
except ZeroDivisionError:
  print("Divided by zero")
finally:
  print("This code will run no matter what")
```

'try:' part will always run first, and then you have two situation:

1. If "try" part executed correctly, "except:" part will be ignored
2. If any error in "try:" part, it will still run until error happened. when error happened, the rest of "try:" part will be ignored, and then jump into "except:" part immediately.

No matter what happened(at both situations), the "finally" part always run. (even some error also happened in 'except' part...)

Raising Exceptions

You can raise exceptions by using the raise statement.

print(1)
raise ValueError
print(2)

Result:>>>
1
ValueError
>>>
You need to specify the type of the exception raised.

In except blocks, the raise statement can be used without arguments to re-raise whatever exception occurred. For example:

```
try:
  num = 5 / 0
except:
  print("An error occurred")
  raise
```

Result:>>>
An error occurred

ZeroDivisionError: division by zero
>>>

Assertions

An assertion is a sanity-check that you can turn on or turn off when you have finished testing the program. An expression is tested, and if the result comes up false, an exception is raised. Assertions are carried out through the use of the assert statement.

```
print(1)
assert 2 + 2 == 4
print(2)
assert 1 + 1 == 3
print(3)
```

Result >>>

```
1
2
AssertionError
>>>
```

Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

The assert can take a second argument that is passed to the AssertionError raised if the assertion fails.

```
temp = -10
assert (temp >= 0), "Colder than absolute zero!"
```

AssertionError exceptions can be caught and handled like any other exception using the try-except statement, but if not handled, this type of exception will terminate the program.

Opening Files

You can use Python to read and write the contents of files. Text files are the easiest to manipulate. Before a file can be edited, it must be opened, using the open function.

```
myfile = open("filename.txt")
```

The argument of the open function is the path to the file. If the file is in the current working directory of the program, you can specify only its name.

You can specify the mode used to open a file by applying a second argument to the open function.

Sending "r" means open in read mode, which is the default.

Sending "w" means write mode, for rewriting the contents of a file.

Sending "a" means append mode, for adding new content to the end of the file.

Adding "b" to a mode opens it in binary mode, which is used for non-text files (such as image and sound files).

For example:

```
# write mode
open("filename.txt", "w")

# read mode
open("filename.txt", "r")
open("filename.txt")

# binary write mode
open("filename.txt", "wb")
```

You can use the + sign with each of the modes above to give them extra access to files. For example, r+ opens the file for both reading and writing.

Once a file has been opened and used, you should close it. This is done with the close method of the file

```
object.file = open("filename.txt", "w")
# do stuff to the file
file.close()
```

Reading Files

To read only a certain amount of a file, you can provide a number as an argument to the read function. This determines the number of bytes that should be read. You can make more calls to read on the same file object to read more of the file byte by byte. With no argument, read returns the rest of the file.file = open("filename.txt", "r")

```
print(file.read(16))
print(file.read(4))
print(file.read(4))
print(file.read())
file.close()
```

After all the contents in a file has been read, any attempts to read further from that file will return an empty string, because you are trying to read from the end of the file.

```
file = open("filename.txt", "r")
file.read()
print("Re-reading")
print(file.read())
print("Finished")
file.close()
```

Just like passing no arguments, negative values will return the entire contents.

To retrieve each line in a file, you can use the readlines method to return a list in which each element is a line in the file. For example:

```
file = open("filename.txt", "r")
print(file.readlines())
file.close()
```

You can also use a for loop to iterate through the lines in the file:

```
file = open("filename.txt", "r")
for line in file:
   print(line)
file.close()
```

In the output, the lines are separated by blank lines, as the print function automatically adds a new line at the end of its output.

The "w" mode will create a file, if it does not already exist.

When a file is opened in write mode, the file's existing content is deleted.

```
file = open("newfile.txt", "r")
print("Reading initial contents")
print(file.read())
print("Finished")
file.close()

file = open("newfile.txt", "w")
file.write("Some new text")
file.close()

file = open("newfile.txt", "r")
print("Reading new contents")
print(file.read())
print("Finished")
file.close()
```

The write method returns the number of bytes written to a file, if successful.

It is good practice to avoid wasting resources by making sure that files are always closed after they have been used. One way of doing this is to use try and finally.

```
try:
  f = open("filename.txt")
  print(f.read())
finally:
  f.close()
```

This ensures that the file is always closed, even if an error occurs.

An alternative way of doing this is using with statements. This creates a temporary variable(often called f), which is only accessible in the indented block of the with statement.

```
with open("filename.txt") as f:
  print(f.read())
```

The file is automatically closed at the end of the with statement, even if exceptions occur within it.

None

The None object is used to represent the absence of a value. It is similar to null in other programming languages. Like other "empty" values, such as 0, [] and the empty string, it is False when converted to a Boolean variable. When entered at the Python console, it is displayed as the empty string.

```
>>> None == None
True
>>> None
>>> print(None)
None
>>>
```

The None object is returned by any function that doesn't explicitly return anything else.

```
def some_func():
 print("Hi!")
var = some_func()
print(var)
```

Dictionaries

Dictionaries are data structures used to map arbitrary keys to values. Lists can be thought of as dictionaries with integer keys within a certain range. Dictionaries can be indexed in the same way as lists, using square brackets containing keys. Example:

```
ages = {"Dave": 24, "Mary": 42, "John": 58}
print(ages["Dave"])
print(ages["Mary"])
```

Each element in a dictionary is represented by a key:value pair.

Trying to index a key that isn't part of the dictionary returns a KeyError.

Example:

```
primary = {
 "red": [255, 0, 0],
 "green": [0, 255, 0],
 "blue": [0, 0, 255],
}


print(primary["red"])
print(primary["yellow"])
```

An empty dictionary is defined as {}.

Only immutable objects can be used as keys to dictionaries. Immutable objects are those that can't be changed. So far, the only mutable objects you've come across are lists and dictionaries. Trying to use a mutable object as a dictionary key causes a TypeError.

```
bad_dict = {

  [1, 2, 3]: "one two three",

}
```

Just like lists, dictionary keys can be assigned to different values. However, unlike lists, a new dictionary key can also be assigned a value, not just ones that already exist.

```
squares = {1: 1, 2: 4, 3: "error", 4: 16,}
squares[8] = 64
squares[3] = 9
print(squares)
```

To determine whether a key is in a dictionary, you can use in and not in, just as you can for a list.

```
nums = {
 1: "one",
 2: "two",
 3: "three",
}
print(1 in nums)
print("three" in nums)
print(4 not in nums)
```

A useful dictionary method is get. It does the same thing as indexing, but if the key is not found in the dictionary it returns another specified value instead ('None', by default). Example:

```
pairs = {1: "apple",
 "orange": [2, 3, 4],
 True: False,
 None: "True",
}
print(pairs.get("orange"))
print(pairs.get(7))
print(pairs.get(12345, "not in dictionary"))
```

```
Output
[2, 3, 4]
None
not in dictionary
```

Tuples

Tuples are very similar to lists, except that they are immutable (they cannot be changed). Also, they are created using parentheses, rather than square brackets. Example:

```
words = ("spam", "eggs", "sausages",)
```

You can access the values in the tuple with their index, just as you did with lists:

print(words[0])

Trying to reassign a value in a tuple causes a TypeError.

words[1] = "cheese"

Like lists and dictionaries, tuples can be nested within each other.

Tuples can be created without the parentheses, just by separating the values with commas. Example:

my_tuple = "one", "two", "three"

print(my_tuple[0])

An empty tuple is created using an empty parenthesis pair.

tpl = ()

Tuples are faster than lists, but they cannot be changed.

List Slices

List slices provide a more advanced way of retrieving values from a list. Basic list slicing involves indexing a list with two colon-separated integers. This returns a new list containing all the values in the old list between the indices. Example:

squares = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

print(squares[2:6])

print(squares[3:8])

print(squares[0:1])

Like the arguments to range, the first index provided in a slice is included in the result, but the second isn't.

If the first number in a slice is omitted, it is taken to be the start of the list. If the second number is omitted, it is taken to be the end. Example:

squares = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

print(squares[:7])

print(squares[7:])

Slicing can also be done on tuples.

List slices can also have a third number, representing the step, to include only alternate values in the slice.

squares = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
print(squares[::2])
print(squares[2:8:3])

Negative values can be used in list slicing (and normal list indexing). When negative values are used for the first and second values in a slice (or a normal index), they count from the end of the list.
squares = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
print(squares[1:-1])

Result:>>>
[1, 4, 9, 16, 25, 36, 49, 64]
>>>
If a negative value is used for the step, the slice is done backwards. Using [::-1] as a slice is a common and idiomatic way to reverse a list.

List Comprehensions

List comprehensions are a useful way of quickly creating lists whose contents obey a simple rule. For example, we can do the following:
 # a list comprehension
*cubes = [i**3 for i in range(5)]*
print(cubes)

Result:>>>
[0, 1, 8, 27, 64]
>>>
List comprehensions are inspired by set-builder notation in mathematics.

A list comprehension can also contain an if statement to enforce a condition on values in the list. Example:
*evens=[i**2 for i in range(10) if i**2 % 2 == 0]*
print(evens)

Trying to create a list in a very extensive range will result in a MemoryError.

This code shows an example where the list comprehension runs out of memory.

*even = [2*i for i in range(10**100)]*

String Formatting

So far, to combine strings and non-strings, you've converted the non-strings to strings and added them.

String formatting provides a more powerful way to embed non-strings within strings. String formatting uses a string's format method to substitute a number of arguments in the string. Example:

 # string formatting

nums = [4, 5, 6]

msg = "Numbers: {0} {1} {2}". format(nums[0], nums[1], nums[2])

print(msg)

Each argument of the format function is placed in the string at the corresponding position, which is determined using the curly braces { }.

print("Student names : {0} {1} {2} {3}".format("Diwash Sapkota", "Ashish Basnet", "Mahesh S Madai", "Babish Shrestha"))

Result:

Student names : Diwash Sapkota Ashish Basnet Mahesh S Madai Babish Shrestha

String formatting can also be done with named arguments. Example:

a = "{x}, {y}".format(x=5, y=12)

print(a)

String Functions

Python contains many useful built-in functions and methods to accomplish common tasks.

join - joins a list of strings with another string as a separator.

replace - replaces one substring in a string with another.

startswith and endswith - determine if there is a substring at the start and end of a string, respectively.

To change the case of a string, you can use lower and upper.

The method split is the opposite of join, turning a string with a certain separator into a list.

Some examples:

```
print(", ".join(["spam", "eggs", "ham"]))
#prints "spam, eggs, ham"

print("Hello ME".replace("ME", "world"))
#prints "Hello world"

print("This is a sentence.".startswith("This"))
# prints "True"

print("This is a sentence.".endswith("sentence."))
# prints "True"

print("This is a sentence.".upper())
# prints "THIS IS A SENTENCE."

print("AN ALL CAPS SENTENCE".lower())
#prints "an all caps sentence"

print("spam, eggs, ham".split(", "))
#prints "['spam', 'eggs', 'ham']"
```

Numeric Functions

To find the maximum or minimum of some numbers or a list, you can use max or min. To find the distance of a number from zero (its absolute value), use abs. To round a number to

a certain number of decimal places, use round. To find the total of a list, use sum. Some examples:

```
print(min(1, 2, 3, 4, 0, 2, 1))
print(max([1, 4, 9, 2, 5, 6, 8]))
print(abs(-99))
print(abs(42))
print(sum([1, 2, 3, 4, 5]))
```

List Functions

Often used in conditional statements, all and any take a list as an argument, and return True if all or any (respectively) of their arguments evaluate to True (and False otherwise). The function enumerate can be used to iterate through the values and indices of a list simultaneously.

Example:

```
nums = [55, 44, 33, 22, 11]
if all([i > 5 for i in nums]):
  print("All larger than 5")
if any([i % 2 == 0 for i in nums]):
  print("At least one is even")
for v in enumerate(nums):
  print(v)
```

Result:>>>

```
All larger than 5
At least one is even
(0, 55)
(1, 44)
(2, 33)
(3, 22)
(4, 11)
>>>
```

Text Analyzer

This is an example project, showing a program that analyzes a sample file to find what percentage of the text each character occupies. This section shows how a file could be open and read.

```
filename = input("Enter a filename: ")
with open(filename) as f:
  text = f.read()
print(text)
```

This part of the program shows a function that counts how many times a character occurs in a string.

```
def count_char(text, char):
 count = 0
 for c in text:
   if c == char:
     count += 1
 return count
```

This function takes as its arguments the text of the file and one character, returning the number of times that character appears in the text. Now we can call it for our file.

```
filename = input("Enter a filename: ")
with open(filename) as f:
 text = f.read()
print(count_char(text, "r"))
```

The next part of the program finds what percentage of the text each character of the alphabet occupies.

```
for char in "abcdefghijklmnopqrstuvwxyz":
 perc = 100 * count_char(text, char) / len(text)
 print("{0} - {1}%".format(char, round(perc, 2)))
```

Let's put it all together and run the program:

```
def count_char(text, char):
 count = 0
 for c in text:
   if c == char:
     count += 1
 return count

filename = input("Enter a filename: ")
with open(filename) as f:
 text = f.read()

for char in "abcdefghijklmnopqrstuvwxyz":
 perc = 100 * count_char(text, char) / len(text)
 print("{0} - {1}%".format(char, round(perc, 2)))
```

Functional Programming

Functional programming is a style of programming that (as the name suggests) is based around functions. A key part of functional programming is higher-order functions. We have seen this idea briefly in the previous lesson on functions as objects. Higher-order functions take other functions as arguments, or return them as results. Example:

```
def apply_twice(func, arg):
  return func(func(arg))

def add_five(x):
  return x + 5
print(apply_twice(add_five, 10))
```

Pure Functions

Functional programming seeks to use pure functions. Pure functions have no side effects, and return a value that depends only on their arguments. This is how functions in math work: for example, The cos(x) will, for the same value of x, always return the same result. Below are examples of pure and impure functions.

Pure function:

```
def pure_function(x, y):
 temp = x + 2*y
 return temp / (2*x + y)
```

Impure function:

```
some_list = []
def impure(arg):
 some_list.append(arg)
```

Using pure functions has both advantages and disadvantages. Pure functions are:

- easier to reason about and test.
- more efficient. Once the function has been evaluated for an input, the result can be stored and referred to the next time the function of that input is needed, reducing the number of times the function is called. This is called memoization.
- easier to run in parallel.

The main disadvantage of using only pure functions is that they majorly complicate the otherwise simple task of I/O, since this appears to inherently require side effects. They can also be more difficult to write in some situations.

Lambdas

Creating a function normally (using def) assigns it to a variable automatically. This is different from the creation of other objects - such as strings and integers - which can be created on the fly, without assigning them to a variable. The same is possible with functions, provided that they are created using lambda syntax. Functions created this way are known as anonymous. This approach is most commonly used when passing a simple function as an argument to another function. The syntax is shown in the next example and consists of the lambda keyword followed by a list of arguments, a colon, and the expression to evaluate and return.

```
def my_func(f, arg):
 return f(arg)

my_func(lambda x: 2*x*x, 5)
```

Lambda functions get their name from lambda calculus, which is a model of computation invented by Alonzo Church.

Lambda functions aren't as powerful as named functions. They can only do things that require a single expression - usually equivalent to a single line of code. Example:

```
#named function
def polynomial(x):
   return x**2 + 5*x + 4
print(polynomial(-4))

#lambda
print((lambda x: x**2 + 5*x + 4) (-4))
```

In the code above, we created an anonymous function on the fly and called it with an argument.

Lambda functions can be assigned to variables, and used like normal functions. Example:

```
double = lambda x: x * 2
print(double(7))
```

However, there is rarely a good reason to do this - it is usually better to define a function with def instead.

map

The built-in functions map and filter are very useful higher-order functions that operate on lists (or similar objects called iterables). The function map takes a function and an iterable as arguments, and returns a new iterable with the function applied to each argument. Example:

```
def add_five(x):
  return x + 5

nums = [11, 22, 33, 44, 55]
result = list(map(add_five, nums))
print(result)
```

We could have achieved the same result more easily by using lambda syntax.

```
nums = [11, 22, 33, 44, 55]
result = list(map(lambda x: x+5, nums))
print(result)
```

To convert the result into a list, we used list explicitly.

filter

The function filter filters an iterable by removing items that don't match a predicate (a function that returns a Boolean). Example:

```
nums = [11, 22, 33, 44, 55]
res = list(filter(lambda x: x%2==0, nums))
print(res)
```

Like map, the result has to be explicitly converted to a list if you want to print it.

Generators

Generators are a type of iterable, like lists or tuples. Unlike lists, they don't allow indexing with arbitrary indices, but they can still be iterated through with for loops. They can be created using functions and the yield statement. Example:

```
def countdown():
 i=5
 while i > 0:
   yield i
   i -= 1

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

The yield statement is used to define a generator, replacing the return of a function to provide a result to its caller without destroying local variables.

Due to the fact that they yield one item at a time, generators don't have the memory restrictions of lists. In fact, they can be infinite!

```
def infinite_sevens():
 while True:
   yield 7

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

In short, generators allow you to declare a function that behaves like an iterator, i.e. it can be used in a for loop.

Finite generators can be converted into lists by passing them as arguments to the list function.

```
def numbers(x):
 for i in range(x):
   if i % 2 == 0:
     yield i

print(list(numbers(11)))
```

Using generators results in improved performance, which is the result of the lazy (on demand) generation of values, which translates to lower memory usage. Furthermore, we do not need to wait until all the elements have been generated before we start to use them.

Decorators

Decorators provide a way to modify functions using other functions. This is ideal when you need to extend the functionality of functions that you don't want to modify. Example:

```
def decor(func):
 def wrap():
   print("============")
   func()
   print("============")
 return wrap

def print_text():
 print("Hello world!")

decorated = decor(print_text)
decorated()
```

We defined a function named decor that has a single parameter func. Inside decor, we defined a nested function named wrap. The wrap function will print a string, then call func(), and print another string. The decor function returns the wrap function as its result.

We could say that the variable decorated is a decorated version of print_text - it's print_text plus something.

In fact, if we wrote a useful decorator we might want to replace print_text with the decorated version altogether so we always got our "plus something" version of print_text. This is done by re-assigning the variable that contains our function:
print_text = decor(print_text)
print_text()

In our previous example, we decorated our function by replacing the variable containing the function with a wrapped version.

```
def print_text():
 print("Hello world!")

print_text = decor(print_text)
```

This pattern can be used at any time, to wrap any function. Python provides support to wrap a function in a decorator by pre-pending the function definition with a decorator name and the @ symbol.

If we are defining a function we can "decorate" it with the @ symbol like:

```
@decor
def print_text():
 print("Hello world!")
```

This will have the same result as the above code.
A single function can have multiple decorators.

itertools

The module itertools is a standard library that contains several functions that are useful in functional programming. One type of function it produces is infinite iterators.

The function count counts up infinitely from a value.

The function cycle infinitely iterates through an iterable (for instance a list or string).

The function repeat repeats an object, either infinitely or a specific number of times.

Example:

```
from itertools import count


for i in count(3):
 print(i)
 if i >=11:
   break
```

There are many functions in itertools that operate on iterables, in a similar way to map and filter. Some examples:

takewhile - takes items from an iterable while a predicate function remains true;

chain - combines several iterables into one long one;

accumulate - returns a running total of values in an iterable.

```
from itertools import accumulate, takewhile
nums = list(accumulate(range(8)))
print(nums)
print(list(takewhile(lambda x: x<= 6, nums)))
```

Result:

```
>>>
[0, 1, 3, 6, 10, 15, 21, 28]
[0, 1, 3, 6]
>>>
```

There are also several combinatoric functions in itertool, such as product and permutation. These are used when you want to accomplish a task with all possible combinations of some items. Example:

```
from itertools import product, permutations
letters = ("A", "B")
print(list(product(letters, range(2))))
print(list(permutations(letters)))
```

OBJECT ORIENTED PROGRAMMING:

Classes

We have previously looked at two paradigms of programming - imperative (using statements, loops, and functions as subroutines), and functional (using pure functions, higher-order functions, and recursion). Another very popular paradigm is object-oriented programming (OOP). Objects are created using classes, which are actually the focal point of OOP. The class describes what the object will be, but is separate from the object itself. In other words, a class can be described as an object's blueprint, description, or definition.

You can use the same class as a blueprint for creating multiple different objects. Classes are created using the keyword class and an indented block, which contains class methods (which are functions). Below is an example of a simple class and its objects.

```
class Cat:
 def __init__(self, color, legs):
   self.color = color
   self.legs = legs

felix = Cat("ginger", 4)
rover = Cat("dog-colored", 4)
stumpy = Cat("brown", 3)
```

This code defines a class named Cat, which has two attributes: color and legs. Then the class is used to create 3 separate objects of that class.

__init__

The __init__ method is the most important method in a class. This is called when an instance (object) of the class is created, using the class name as a function. All methods

must have self as their first parameter, although it isn't explicitly passed, Python adds the self argument to the list for you; you do not need to include it when you call the methods. Within a method definition, self refers to the instance calling the method.

Instances of a class have attributes, which are pieces of data associated with them. In this example, Cat instances have attributes color and legs. These can be accessed by putting a dot, and the attribute name after an instance. In an __init__ method, self.attribute can therefore be used to set the initial value of an instance's attributes. Example:

```
class Cat:
 def __init__(self, color, legs):
   self.color = color
   self.legs = legs

felix = Cat("ginger", 4)
print(felix.color)
```

In the example above, the __init__ method takes two arguments and assigns them to the object's attributes. The __init__ method is called the class constructor.

Methods

Classes can have other methods defined to add functionality to them. Remember, that all methods must have self as their first parameter. These methods are accessed using the same dot syntax as attributes. Example:

```
class Dog:
 def __init__(self, name, color):
   self.name = name
   self.color = color
 def bark(self):
   print("Woof!")

fido = Dog("Fido", "brown")
print(fido.name)
fido.bark()
```

Classes can also have class attributes, created by assigning variables within the body of the class. These can be accessed either from instances of the class, or the class itself. Example:

```
class Dog:
 legs = 4
 def __init__(self, name, color):
   self.name = name
   self.color = color

fido = Dog("Fido", "brown")
print(fido.legs)
print(Dog.legs)
```

Trying to access an attribute of an instance that isn't defined causes an AttributeError. This also applies when you call an undefined method. Example:

```
class Rectangle:
 def __init__(self, width, height):
   self.width = width
   self.height = height

rect = Rectangle(7, 8)
print(rect.color)
```

LOOPS

OOP in python (objects, inheritance, polymorphism)

Explore the libraries.

Decide what you plan to do.