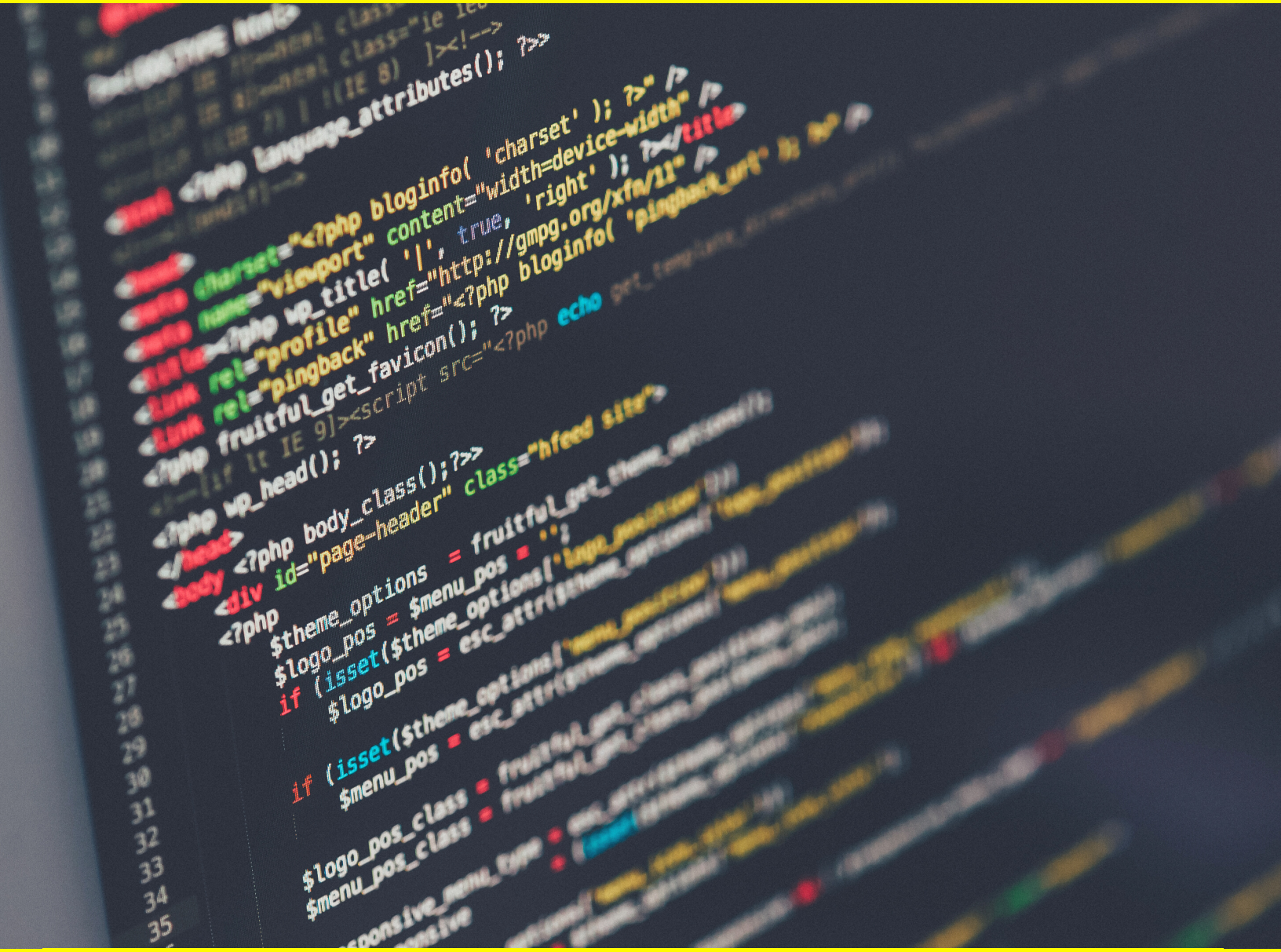
the fast lane to python

PART 1



BY:

Arshadul Shaikh

Ref taken from Coursera + Udemy

Understanding computer programs requires **algorithmic, mechanistic thinking**. Programs specify mechanistic sequences of actions to perform; when executed, they transform input data into output data. They execute very **reliably**, and very **fast**, but **not creatively.** Computers do what you tell them to do, not what you mean for them to do. Thus, understanding computer code involves a lot of mental simulation of what will actually happen, not what you wish would happen.

Writing computer programs requires not only mechanistic thinking but creative problem solving. Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem solving skills.

Algorithms:

Algorithms are like recipes: they must be followed exactly, they must be clear and unambiguous, and they must end.

Our **goal in programming** is to take a problem and develop an algorithm that can serve as a **general solution**. Once we have such a solution, we can express it as a program and **use our computer to automate** the execution. These programs are written **in programming languages.**

Python

Python is a **High-level language** similar to other such languages like C++, Java, etc.

Low-level languages on the contrary are the ones referred to as **machine languages** or assembly languages which **computers can execute**. High level languages have to processed and converted first into machine language which computer can understand before they can run, this extra processing takes some time which is one small disadvantage of high-level languages, however advantages outweigh anytime.

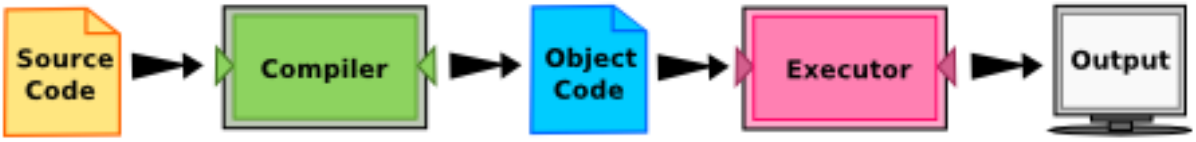
As high-level language is in **human readable** form, it is quite **easier** to write and requires **less time** also they are **shorter** and easier to read, and they are more likely to be correct. Second, high-level languages are **portable**, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can run on only one kind of computer and have to be rewritten to run on another.

Two kinds of programs process high-level languages into low-level languages: **interpreters and compilers.** An interpreter reads a high-level program and executes it, meaning that it does what the program says. It processes the program a line by line and performs computations.

A compiler reads the program and translates it completely before the program starts running. In this case, the high-level program is called the source code, and the translated program is called the object code or the executable. Once a program is compiled, you can execute it repeatedly without further translation.

Many HL languages use both processes. They are first compiled into a LL language, called bytecode, and then interpreted by a program called a virtual machine(like JVM).

Python uses both processes, but because of the way programmers interact with it, it is usually considered an **interpreted** language.



**If the instructions are stored in a file, it is called the source code file.

FUN FACTS:

I hope no-one thought that High-level language is high level if you are standing and low-level if you are sitting :D

Program:

Program is a sequence of instructions that specifies how to perform a computation. The computation could be anything like rendering an html page in web browser or encoding a video and streaming it across the network.

It is just step-by-step instructions that the computer can understand and execute. Programs often implement algorithms, but note that algorithms are typically less precise than programs and do not have to be written in a programming language.

Some of the generic terms are:

input: Get data from the keyboard, a file, or some other device.
output: Display data on the screen or send data to a file or other device.
math and logic: Perform basic mathematical operations like addition and multiplication and logical operations like and, or, and not.
conditional execution: Check for certain conditions and execute the appropriate sequence of statements.
repetition: Perform some action repeatedly, usually with some variation.

This is all every program is made of. Thus a programming is a process of breaking a large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed with sequences.

Formal and Natural Languages:

Natural languages are the languages that people speak, such as English, Spanish, Hindi. They were not designed by people (although people try to impose some order to them) they evolved naturally.

Formal Languages are languages that are designed by people for specific applications. For eg. the notation that mathematicians use a formal language to denote relationship among numbers and symbols, similarly Chemists use a formal language to represent the chemical structure of molecules.

> ***Programming languages are formal languages that have been designed to express computations.***

Formal language tend to have strict rules about syntax. For eg. **3+3=6** is syntactically **correct** mathematical statement, but **3+=$5** is **incorrect**.

Syntax rules come in two flavors, pertaining to **tokens** and **structures**. Tokens are the basic elements of a language such as words, numbers and chemical elements. above example of mathematical expression was wrong because **$** i**s not a legal** token for mathematics.

Structure is how the tokens are arranged . So the above example was also structurally illegal as we do not put a logical operator next to addition sign.

Although formal and natural languages have many features in common — tokens, structure, syntax, and semantics — there are many differences:

ambiguity: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy: In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

literalness: Formal languages mean exactly what they say. On the other hand, natural languages are full of idiom and metaphor. If someone says, “The other shoe fell”, there is probably no shoe and nothing falling.

Typical First Program:

Traditionally, the first program written in a new language is called Hello, World! because all it does is display the words, Hello, World!

In Python, the source code looks like this:

```
In [6]: print ('Hello, World')

        Hello, World
```

This is an example of print function, printing anything that you give between those brackets.

Points to learn from this small program :

- you don't have to write big code lines like cout<<"hello, world"; in cpp or System.out.println('hello, world") in java , you can put anything in print() with single or double inverted commas and you are good to go (will study in detail about what is **print**)

- Also like Unix based scripting languages you can write down any comment using "#" as python interpreter ignores anything that comes after hash.

- We would need a proper closure of braces () for not getting any compilation error ("hello, world" will give an error .

- The quotation marks in the program mark the beginning and end of the value. They don't appear in the result.

*****KEY POINTS:**

- To check your **understanding** about the state of variables before your code snippet runs, add diagnostic **print** statements as much as you want which print out the types and values of variables you are using in expressions. Try to add these print statements just before the code snippet you are trying to understand.

- The diagnostic print statements which you added above are temporary. Once you have verified that a program is doing what you think it's doing, you will remove these extra print statements.

Comments:

Formal languages are **dense & complicated**, they get more difficult to read as we start writing bigger programs, in order to avoid getting lost in the code it is better to **add notes** wherever possible in natural language which will help the reader to **understand** what program is doing. These are called comments.

A **comment** in a computer program is text that is intended only for the human reader - it is completely ignored by the interpreter. In Python, the **# token** starts a comment.

In [8]:
```
#------------------------------------------------
# This demo program shows off how elegant Python is!
#------------------------------------------------

print("Hello, World!")     # Isn't this easy!
```

```
Hello, World!
```

Glossary:

debugging: The process of finding and removing any of the three kinds of programming errors: ***syntax error*, *semantic error*, and *runtime error*.**

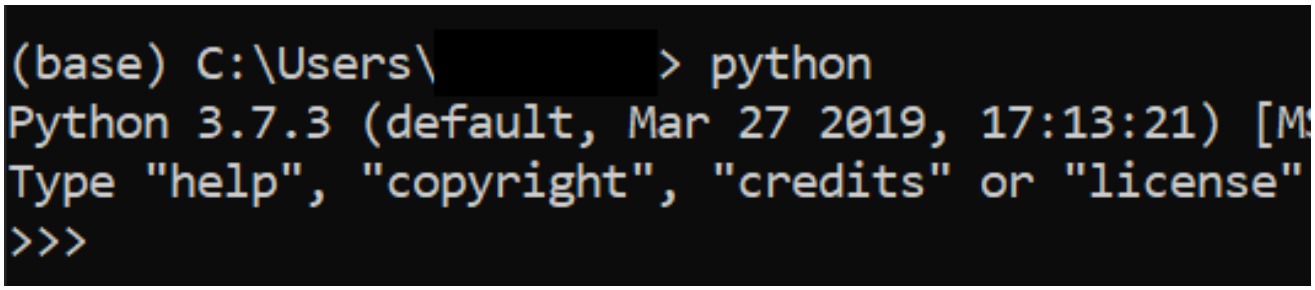
Python shell: An interactive user interface to the Python interpreter, and the user of a Python shell types commands at the prompt (>>>), and presses the return key to send these commands immediately to the interpreter for processing. To initiate the Python Shell, the user should open the terminal and type “python”. Once the user presses enter, the Python Shell appears and the user can interact with it.

runtime error: An error that does not occur until the program has started to execute but that prevents the program from continuing.

semantic error: An error in a program that makes it do something other than what the programmer intended.

semantics: The meaning of a program

shell mode: A mode of using Python where expressions can be typed and executed in the command prompt, and the results are shown immediately in the command terminal window. Shell mode is initiated by opening the terminal of your operating system and typing “python”. Press enter and the Python Shell will appear. This is in contrast to source code. Also see the entry under Python shell.



syntax: The structure of a program.

syntax error: An error in a program that makes it impossible to parse — and therefore impossible to interpret.

Introduction to basics:

Everything has some basic building blocks behind, that are assembled together to showcase what you see with your naked eyes, —from the software running on your smartwatch, to the infrastructure behind the largest websites, and every app running on your phone, they all were made by using some basic building blocks. Let's try to see what python has for us:

Value and Data types:

- A **value** is one of the **fundamental things** — like a word or a number — that a program manipulates. Example of values could be 5 or "Hello, World!".

- These objects are classified into different classes, or data types: 5 is an integer, and “Hello, World!” is a string as it contains a sequence of letters.
- You (and the interpreter) can identify strings because they are enclosed in quotation marks. In the same way, we can specify values directly in the programs we write.
- For example we can specify a **number as a literal** just by (literally) **typing** it **directly** into the program (e.g., 5 or 4.32). In a program, we specify a word, or more generally a string of characters, by enclosing the characters inside quotation marks (e.g., "Hello, World!").
- During execution of a program, the Python interpreter creates an internal representation of literals that are specified in a program. It can then manipulate them, for example by multiplying two numbers.
- We call the internal representations objects or just values.

- You can’t directly see the internal representations of values python interpreter has but you can however, use the **print** function to see a printed representation in the output window.
- The printed representation of a character string, however, is not exactly the same as the literal used to specify the string in a program. For the literal in a program, you enclose the string in quotation marks. The printed representation**, in the output window, omits the quotation marks.**

```
In [13]: print(4.32)
         print("Hello,World!")
         print (5)
```

```
4.32
Hello,World!
5
```

- Numbers with a decimal point belong to a class called float, because these numbers are represented in a format called floating-point.

*****KEY POINT:**
Python turns literals into values which have internal representation that users never get to see directly. Outputs are external representation of values that appear in the output window.

Operators and Operands:

You can **build complex expressions** out of simpler ones using **operators**. Operators are **special tokens** that represent **computations** like addition, multiplication and division. The values the operator works on are called **operands**.

- + is addition Operator
- - is subtraction Operator
- * is a token for Multiplication
- () use of parentheses for grouping
- ** is the token for exponentiation
- / is the division operator which produces a floating point result (even if the result is an integer; 4/2 is 2.0). If you want truncated division, which ignores the remainder, you can use the // operator (for example, 5//2 is 2).
- % is the modulus operator/remainder Operator works on int (int expressions) and yields the remainder.

All these operators mean in python the same what they mean in mathematics. Remember that if we want to see results of the computation, the program needs to specify with **print**

```
In [15]: 20 + 32
         5 ** 2
         (5 + 9) * (15 - 7)
         print(7 + 5)
         print(9 / 5)
         print(5 / 9)
         print(9 // 5)
```

```
12
1.8
0.5555555555555556
1
```

*****STAR POINTS:**

- The truncated division operator, //, also works on floating point numbers. It truncates to the nearest integer, but still produces a floating point result. Thus 7.0 // 3.0 is 2.0.
- The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another —if x % y is zero, then x is divisible by y. Also, you can extract the right-most digit or digits from a number. For example, x % 10 yields the right-most digit of x (in base 10). Similarly x % 100 yields the last two digits.

Function Calls:

You might remember high school Algebra where we defined a function f by specifying how it transformed an input into an output, f(x)=3x + 2. Then we wrote f(5) and expected the output value to be 17.

Python too can compute new values with function calls. It adopts the same syntax for invoking functions. There are several built-in functions available in Python which we will be using to make our life and understanding of python easy.

Functions can be considered as a factor where you feed some material , and let the defined operation perform the work and return you an object.

***** STAR POINT:**
print that you have read so far is actually a function. All functions produce output but only print() causes things to appear on screen.

we will be defining several functions , below are some examples of how we can define a function and then use it immediately

In [23]:
```
def square(x):
    return x * x

def sub(x, y):
    return x - y

print(square(3))
square(5)
print(sub(6, 4))
print(sub(5, 9))
```

```
9
2
-4
```

- square function aboves is taking a single input parameter, and should return input multiplied by itself.
- sub function takes two input parameters and returns the result of subtracting the second from the first.
- when a function takes more than one input parameter, the inputs are separated by a comma.
- Also notice that the order of the inputs matters. The value before the comma is treated as the first input, the value after it as the second input.

Function calls can also be done as part of complex expressions:

In [24]:
```
def square(x):
    return x * x

def sub(x, y):
    return x - y

print(square(3) + 2)
print(sub(square(3), square(1+1)))
```

```
11
5
```

Here you need to understand that we are taking square(), sub() function as literal just like we used numbers earlier.

- Notice that we always have to resolve the expression inside the innermost parentheses first , in order to determine what input to provide when calling the functions.

    - print(sub(square(3), square(1+1)))
    - print(sub(9, square(1+1)))
    - print(sub(9, square(2)))
    - print(sub(9, 4))
    - print(5)

Functions are objects; parentheses invoke functions:

Functions are themselves just objects, if you give function name to print without parentheses , you will probably get not so good printed representation telling you that the given name inside print is a function.

Just typing the name of the function refers to the function as an object. Typing the name of the function followed by parentheses () invokes the function.

In [25]:
```
def square(x):
    return x * x

print(square)
print(square(3))
```

```
<function square at 0x00000191CF3B4510>
9
```

Data Types:

If you are not sure what **class(data type)** your types value is , python provides you with a function called **type ,** which comes handy.

27]:
```
print(type("Hello, World!"))
print(type(17))
print("Hello, World")
print(type(3.2))
print(type("17"))
print(type("3.2"))
```

```
<class 'str'>
<class 'int'>
Hello, World
<class 'float'>
<class 'str'>
<class 'str'>
```

But if you see the values 17 & 3.2 are also considered as str class???? **Because** they are inside quotation marks and anything inside "" is considered as a string by the python interpreter.

Strings in Python can be enclosed in either **single quotes** (') or **double quotes** ("), or **three of each (''' or """)**

In [28]:
```
print(type('This is a string.'))
print(type("And so is this."))
print(type("""and this."""))
print(type('''and even this...'''))
```

```
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
```

***** **KEY POINTS:**

- Double quoted strings can contain single quotes inside them, as in "Amish's car"
- single quoted strings can have double quotes inside them, as in 'The knights never sleep "Sleep!"'.
- Strings enclosed with **three occurrences** of either quote symbol are called **triple quoted strings**. They can contain either single or double quotes.

```
In [37]: print('''"Oh no", she exclaimed, "Alexandar's bike is broken!"''')
         print (""" hi , I am "Alexandar the great" and i like 'you', do you """)

"Oh no", she exclaimed, "Alexandar's bike is broken!"
 hi , I am "Alexandar the great" and i like 'you', do you
```

- Triple quoted strings can even span multiple lines.

```
In [39]: print (""" hi , I am "Alexandar the great"
         i like 'you',
         do you? """)

 hi , I am "Alexandar the great"
i like 'you',
do you?
```

- Python language designers usually chose to surround their strings by single quotes. What do you think would happen if the string already contained single quotes?

```
In [46]: print ("hi"")
           File "<ipython-input-46-34d6bc22acab>", line 1
             print ("hi"")
                        ^
         SyntaxError: EOL while scanning string literal
```

When you type a large integer, you might be tempted to use commas between groups of three digits, as in 10 Million would be written as 10,000,000 . This is not a legal integer in Python, but it does mean something else, which is legal:

```
In [47]: print(10000000)
         print(10,000,000)

         10000000
         10 0 0
```

Well, that's not what we expected at all! Because of the **comma**, Python chose to treat this as a **pair of values.** In fact, a **print statement** can print **any number** of values as long as you **separate** them by **commas**. Notice that the values are separated by **spaces** when they are displayed.

```
In [48]: print(42, 17, 56, 34, 11, 4.35, 32)
         print(3.4, "hello", 45)

         42 17 56 34 11 4.35 32
         3.4 hello 45
```

****** KEY POINT**

Remember not to put commas or spaces in your integers, no matter how big they are.

Type conversion functions:

While writing programs in future we might need to **convert** values from one **type** to another. Python provides a few simple functions that will allow us to do that. The functions int, float and str will (attempt to) convert their arguments into types int, float and str respectively. We call these type **conversion functions**.The int function can take a floating point number or a string, and turn it into an int. For floating point numbers, it discards the decimal portion of the number - a process we call truncation towards zero on the number line.
Let us see this in action:

```
print(3.14, int(3.14))
print (3.14, int(3.14))
print(3.9999, int(3.9999))       # This doesn't round to the closest int!
print(3.0, int(3.0))
print(-3.999, int(-3.999))        # Note that the result is closer to zero
print("2345", int("2345"))        # parse a string to produce an int
print(17, int(17))                # int even works on integers
print(int("23bottles"))           #illegal conversion from string to int
```

```
3.14 3
3.14 3
3.9999 3
3.999 3
3.0 3
-3.999 -3
-3.999 3
2345 2345
17 17

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-50-368a594769b6> in <module>
      8 print("2345", int("2345"))        # parse a string to produce an int
      9 print(17, int(17))                # int even works on integers
---> 10 print(int("23bottles"))

ValueError: invalid literal for int() with base 10: '23bottles'
```

Type conversion functions:

While writing programs in future we might need to **convert** values from one **type** to another. Python provides a few simple functions that will allow us to do that. The functions int, float and str will (attempt to) **convert their arguments into types** int, float and str respectively. We call these type **conversion functions**.The int function can take a floating point number or a string, and turn it into an int. For floating point numbers, it discards the decimal portion of the number - a process we call truncation towards zero on the number line.
Let us see this in action:

```
print(3.14, int(3.14))
print (3.14, int(3.14))
print(3.9999, int(3.9999))       # This doesn't round to the closest int!
print(3.0, int(3.0))
print(-3.999, int(-3.999))        # Note that the result is closer to zero
print("2345", int("2345"))        # parse a string to produce an int
print(17, int(17))                # int even works on integers
print(int("23bottles"))           #illegal conversion from string to int
```

```
3.14 3
3.14 3
3.9999 3
3.999 3
3.0 3
-3.999 -3
-3.999 3
2345 2345
17 17

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-50-368a594769b6> in <module>
      8 print("2345", int("2345"))        # parse a string to produce an int
      9 print(17, int(17))                # int even works on integers
---> 10 print(int("23bottles"))

ValueError: invalid literal for int() with base 10: '23bottles'
```

One common operation where you might need to do a type conversion is when you are concatenating several strings together but want to include a numeric value as part of the final string.
Because we can't concatenate a string with an integer or floating point number, we will often have to convert numbers to strings before concatenating them.

```
val=50+5

print ("the value is" + val)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-58-5a4230473907> in <module>
      1 val=50+5
      2
----> 3 print ("the value is" + val)

TypeError: can only concatenate str (not "int") to str
```

The call here failed as we tried to concatenate (+ here is the sign for concatenation) a string with an integer value.
To have a successful output we either need to convert the int value to string or write separately by segregating with a comma as we all know print() function prints everything that we give in the parentheses separated with a comma.

```
:  val=50+5

   print ("the value is",val)
   print ("the value is " + str(val))
```

```
the value is 55
the value is 55
```

***** KEY POINT:**
we remember when we give several expressions to print separated by a comma, print function adds a space after every expression is printed which happened in the first output where the expression was "the value is", val with no space but still we have a space in front of val.

Variables:

What will you do if there is an expression that needs to be used more than once in your program???
If you keep typing the same big expression everywhere then probably you have not understood the concept of **efficient programming**. And to save you from this, programming language gives you the ability to manipulate **variables.**

Assignment statements create new variables and also give them values to refer to.

```
message = "What's up, Doc?"
n = 17
pi = 3.14159

print(message,n,pi)
```

```
What's up, Doc? 17 3.14159
```

In above examples we have done 3 assignments. message variable was assigned with a statement i.e message became a string type variable, while n with an integer value became integer type variable and pi became float type variable.

The assignment token (=) should not be confused with equality. The assignment operator links a name on the left hand side of the operator with a value on the right hand side.

```
5 = var
```

```
  File "<ipython-input-69-0c03b79a4024>", line 1
    5 = var
      ^
SyntaxError: can't assign to literal
```

we have error because?? because as per the interpreter we are trying to assign here expression var to a number which is illegal.

- while reading or writing code, try to say to yourself var is assigned value 5 , or var gets the value 5 ie. var=5 which is the correct way of assignment.

- Variables in a program are meant to remember things, but as the name defines it can change over time for eg. if you want to make a calendar, you cannot keep assigning variable for everyday, you would need some variables to continuously change with days.

- Same variable will be assigned to different value on different days to get the output.

*****KEY POINT**

Do not relate the programming language variable with what you learned in mathematics. As in Algebra, if you give x the value 3, it cannot change or refer to a different value half way through your calculations.

Type casting:

```
day='Monday'
date=21
year=2020
print(day + " " + str(date)+" "+str(year))
day='Tuesday'
date=22
print(day + " " + str(date)+" "+str(year))
day='Wednesday'
date=23
print(day + " " + str(date)+" "+str(year))
```

```
Monday 21 2020
Tuesday 22 2020
Wednesday 23 2020
```

we can find out the data type of the current value of a variable by putting the variable name inside the parentheses following the function name type.

```
day='Monday'
date=21
year=2020
print(type(day))
print(type(date))
print(type(year))
```

```
<class 'str'>
<class 'int'>
<class 'int'>
```

*****KEY POINT:**

If you have programmed in another language such as Java or C++, you may be used to the idea that variables have types that are declared when the variable name is first introduced in a program. Python doesn't do that. Variables don't have types in Python; values do. That means that it is acceptable in Python to have a variable name refer to an integer and later have the same variable name refer to a string.

Variable Names and Keywords:

Variable names can be arbitrarily long. Important things to remember while considering a variable name:

- Can **contain** both letters & digits
- Should **always begin** with a **letter** or an **underscore**
- Though it is **legal** to use **uppercase** letter, by **convention** you **shouldn't** use.
- Variables are **case sensitive** 'python' & 'Python' are 2 different variables.
- Variable can **never contain space**, if you want to have variable to consist multiple words use underscores. For eg. 'python_v1'

```
python1='small letters'
Python1='Capital letters'
_python='with underscore'
print(python,Python,_python)
```

```
small letters Capital letters with underscore
```

- Below example starting with a number is illegal

```
1vPython = "first version python"
```

```
  File "<ipython-input-90-7770735a487c>", line 1
    1vPython = "first version python"
           ^
SyntaxError: invalid syntax
```

- more$ is illegal as it contains special character "$".

```
more$ = 100
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-92-00acba39cf22> in <module>
----> 1 get_ipython().run_line_magic('more', '$ = 100')

~\AppData\Local\Continuum\anaconda3\lib\site-packages\IPython\core\interactiveshell.py in run_line_ma
gic(self, magic_name, line, _stack_depth)
   2305                 kwargs['local_ns'] = sys._getframe(stack_depth).f_locals
   2306             with self.builtin_trap:
-> 2307                 result = fn(*args, **kwargs)
   2308             return result
   2309

<C:\Users\arshaikh\AppData\Local\Continuum\anaconda3\lib\site-packages\decorator.py:decorator-gen-120
> in less(self, arg_s)

~\AppData\Local\Continuum\anaconda3\lib\site-packages\IPython\core\magic.py in <lambda>(f, *a, **k)
    185     # but it's overkill for just that one bit of state.
    186     def magic_deco(arg):
--> 187         call = lambda f, *a, **k: f(*a, **k)
    188
    189         if callable(arg):

~\AppData\Local\Continuum\anaconda3\lib\site-packages\ipykernel\zmqshell.py in less(self, arg_s)
    340             cont = self.shell.pycolorize(openpy.read_py_file(arg_s, skip_encoding_cookie=Fals
e))
    341         else:
--> 342             cont = open(arg_s).read()
    343         page.page(cont)
    344

FileNotFoundError: [Errno 2] No such file or directory: '$ = 100'
```

- But we got error when we assigned the name of the variable as "class" even though it fulfills all conventions , still we get an error of invalid syntax.

```
class = "Python class"
```

```
  File "<ipython-input-93-b6465a1863bf>", line 1
    class = "Python class"
          ^
SyntaxError: invalid syntax
```

It turns out that class is one of **Python keywords.**

Keywords define the language's syntax rules and structure, and they cannot be used as variable names. Python has thirty-something keywords (and every now and again improvements to Python introduce or eliminate one or two):

and	as	assert	break	class	continue
def	del	elif	else	except	exec
finally	for	from	global	if	import
in	is	lambda	nonlocal	not	or
pass	raise	return	try	while	with
yield	True	False	None		

Need to remember these keywords as they will come handy whenever you find a sudden syntax error while assigning the variable name.

***** KEY POINTS:**
You generally would choose names for your variables that are meaningful to the human readers of the program but sometimes you might find funny to use strange or obscene names for your variables. This is not good practice , put this habit of using meaningful names right away.

Statement and Expressions:

A **statement** is an instruction that the python **interpreter** can **understand** and **execute**.

Expression is a **combination** of **literals**, **variable** names, **operators**, and calls to **functions**. Expressions needs to be evaluated and the result of this evaluation is a value or object.

Let's try to take some examples to understand expressions in depth:

```
print((1+2-3+(2*3))//2)
print(len("Hello World!"))
```

```
3
12
```

- In first statement as per BODMAS rule the expression was evaluated and printed.
- 2nd statement , we used len a built-in Python function that returned the number of characters in a string.
- The evaluation of an expression produces a value, which is why expressions can appear on the right hand side of assignment statements. A literal all by itself is a simple expression, and so is a variable.

    - literal (e.g., “Hello” or 3.14)
    - variable name (e.g., x or len)
    - operator expression:
      <expression> operator-name <expression>
    - function call expressions:
      <expression>(<expressions separated by commas>)

***** KEY POINT:**
If you ask Python to **print** an expression, the **interpreter evaluates** the expression and displays the result.

Let's try now to use some user defined functions inside parentheses instead of putting literals and try to solve a complex expression.

```
def square(x):          #this function will multiply the input number with itself
   return x * x

def sub(x, y):          #this will return subtracting first input with second.
   return x - y         #Remember to follow the order

x,y = 2,1               #this form initiatlization is supported by python
print(square(y + 3))
print(square(y + square(x)))
print(sub(square(y), square(x)))
```

```
16
25
-3
```

- It is important to start learning to read code that contains complex expressions. The Python interpreter examines any line of code and parses it into components.
- For example, if it sees an "=" symbol, it will try to treat the whole line as an assignment statement.
    - It will expect to see a valid variable name to the left of the =, and will parse everything to the right of the = as an expression.
    - It will try to figure out whether the right side is a literal, a variable name, an operator expression, or a function call expression.
    - If it’s an operator expression, it will further try to parse the sub-expressions before and after the operator.

- In order to evaluate an operator expression, the Python interpreter first completely evaluates the expression before the operator, then the one after, then combines the two resulting values using the operator.

Let's try to see how a function call expression works if it has sub-expressions :

```
def square(x):
    return x * x

def add(x, y):
    return x + y

x = 5
y = 7
add(square(y), square(x))
```

```
74
```

here we have square function call and also has sub-expression add(square(y),square(x)). Steps done internally to execute the expression are as below:

- **-add-**(square(y), square(x)) -- Firstly the add function was evaluated
- -add-(**-square-(y)**, square(x)) -- square is also a function, so evaluating its arguments in 2nd step.
- -add-(**-square-(7)**, square(x))
- -add-(**49**, square(x)) -- first sub expression which is square function was evaluated
- -add-(49, **-square-(x)**) -- another square function will be evaluated
- -add-(49, **-square-(5)**) -- value substituted
- -add-(49, **25**) -- both function was evaluated
- **74** -- finally add function was passed with arguments

Question:
Please order the code fragments in the order in which the Python interpreter would evaluate them. x is 2 and y is 3. Now the interpreter is executing `square(x + sub(square(y), 2 *x))`.

```
look up the variable square, again, to get the function object
multiply 2 * 2 to get 4
add 2 and 5 to get 7
look up the variable x to get 2
look up the variable x, again, to get 2
look up the variable sub to get the function object
run the square function, again, on input 7, returning the value 49
run the sub function, passing inputs 9 and 4, returning the value 5
run the square function on input 3, returning the value 9
look up the variable y to get 3
look up the variable square to get the function object
```

answer: 11,4,6,1,10,9,5,2,8,3,7

Order of Operations:

When python has more than one operator in an expression, the order of evaluation is dependent on rules of precedence. Python follows the same precedence rules as mathematics :

- **Parentheses** have the **highest precedence** and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first:
    - 2 * (3-1) is 4, and (1+1)**(5-2) is 8. You can also use parentheses to make an expression easier to read, as in (minute * 100) / 60: in this case, the parentheses don't change the result, but they reinforce that the expression in parentheses will be evaluated first.

- **Exponentiation** has the **next highest** precedence, so 2**1+1 is 3 and not 4, and 3*1**3 is 3 and not 27. Can you explain why?
because exponentiation is a non-associative.

- **Multiplication** and both **division** operators have the **same precedence**, which is higher than addition and subtraction, which also have the same precedence. So 2*3-1 yields 5 rather than 4, and 5-2*2 is 1, not 6.
- Operators with the **same precedence** are **evaluated** from **left-to-right**. In algebra we say they are **left-associative**. So in the expression 6-3+2, the subtraction happens first, yielding 3. We then add 2 to get the result 5. If the operations had been evaluated from right to left, the result would have been 6-(3+2), which is 1.

***** KEY POINTS:**

Due to some historical quirk, an **exception** to the **left-to-right left-associative rule** is the exponentiation operator ****.**

```
print(2 ** 3 ** 2)     # the right-most ** operator gets done first!
print((2 ** 3) ** 2)   # use parentheses to force the order you want!
```

```
512
64
```

In order to solve this problems, always remember to use parentheses to force exactly the order you want when exponentiation is involved.

```
16 - 2 * 5 // 3 + 1

#Using parentheses
#the expression is evaluated as (2*5) first
#then (10 // 3)
#then (16-3)
#then (13+1).
```

```
14
```

Reassignment:

it is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

```
a = 5
b = a    # after executing this line, a and b are now equal
print(a,b)
a = 3    # after executing this line, a and b are no longer equal
print(a,b)
```

```
5 5
3 5
```

Line 4 changes the value of a but does not change the value of b, so they are no longer equal. We will have much more to say about equality in a later chapter.

In the first statement a = 5 the literal number 5 evaluates to 5, and is given the name a. In the second statement, the variable a evaluates to 5 and so 5 now ends up with a second name b.

***** KEY POINTS:**
In some programming languages, a different symbol is used for assignment, such as <- or :=. The intent is that this will help to avoid confusion. Python chose to use the tokens = for assignment, and == for equality. This is a popular choice also found in languages like C, C++, Java, and C#.

Updating Variables: (Adv reassignment)

One of the most common forms of reassignment is an update where the new value of the variable depends on the old.

For example: x=x+1
Meaning ? what happened above ? the above expression means get the current value of x, add one, and the update the X with new value.

Although this assignment statement may look a bit strange, remember that executing assignment is a two-step process.

- First, evaluate the right-hand side expression.
- Second, let the variable name on the left-hand side refer to this new resulting object.
- The fact that x appears on both sides does not matter. The semantics of the assignment statement makes sure that there is no confusion as to the result.

```
x = 6          # initialize x
print(x)
x = x + 1      # update x
print(x)
```

```
6
7
```

- you initialized x with 6 and then we updated the value in next lines by adding one and now the value of x became 7.

Updating a **variable** by adding something to it is called an **increment**; **subtracting** is called a **decrement**. Sometimes programmers talk about incrementing or decrementing without specifying by how much; when they do they usually mean by 1. Sometimes programmers also talk about **bumping** a variable, which **means the same** as incrementing it by 1.

In python for incrementing and decrementing we have a special syntax:

- += for incrementing
- -= for decrementing

It is similar to what you would have seen in C/C++ where you would have used "++" and "--"

x+=1 is formal language representation but what it actually means is x=x+1

```
x = 6          # initialize x
print(x)
x += 3         # increment x by 3; same as x = x + 3
print(x)
x -= 1         # decrement x by 1
print(x)
```

```
6
9
8
```

Input:

You have been watching all the examples with values hard-coded so far , wouldn't it be great if you can run the same program again and again and you have the option to provide new inputs all the time to check the efficiency of code.

One way to do this is by using a built-in function called **input** to accomplish this task. The input function allows the programmer to provide a **prompt string.**

If you have used this function in your code, everytime you run the program python interpreter will show you a pop up window asking you to type some text .

Let's see with an example:

```
n = input("Please enter your name: ")  #you can type anything inside the parentheses
print("Hello", n)       #print will consider 2 different strings separated with a comma
print("Hello" +" " +n)  #here we are concatenating Hello with variable n
```

```
Please enter your name: Python
Hello Python
Hello Python
```

***** KEY POINTS:**
It is very important to note that the **input function returns a string value**. Even if you asked the user to enter their age, you would get back a string like "17". It would be your job, as the programmer, to convert that string into an int or a float, using the int or float converter functions we saw earlier.

We often use the word **“input” (or, synonymously, argument)** to **refer** to the **values** that are **passed** to any function. **Do not confuse** that **with** the **input function**, which asks the user of a program to type in a value. The input is a character string that is displayed as a prompt to the user.

The **output is whatever character string the user types**.This is **analogous** to the potential confusion of function **“outputs”** with the **contents** of the **output window**. Every function produces an output, which is a Python value. Only the **print function** puts things in the output window. Most functions take inputs, which are Python values. Only the input function invites users to type something

```
str_seconds = input("Please enter the number of seconds you wish to convert=")
total_secs = int(str_seconds)

hours = total_secs // 3600
secs_still_remaining = total_secs % 3600
minutes =  secs_still_remaining // 60
secs_finally_remaining = secs_still_remaining  % 60

print("Hrs=", hours, "mins=", minutes, "secs=", secs_finally_remaining)
```

```
Please enter the number of seconds you wish to convert=1324
Hrs= 0 mins= 22 secs= 4
```

The variable str_seconds will refer to the string that is entered by the user. As we said above, even though this string may be 7684, it is still a string and not a number. To convert it to an integer, we use the int function. The result is referred to by total_secs. Now, each time you run the program, you can enter a new value for the number of seconds to be converted.

Glossary:

assignment statement
A statement that assigns a value to a name (variable). To the left of the assignment operator, =, is a name. To the right of the assignment token is an expression which is evaluated by the Python interpreter and then assigned to the name

assignment token
"=" is Python's assignment token, which should not be confused with the mathematical comparison operator using the same symbol.

statement
An instruction that the Python interpreter can execute. So far we have only seen the assignment statement, but we will soon meet the import statement and the for statement.

boolean expression
An expression that is either true or false.

boolean value
There are exactly two boolean values: True and False. Boolean values result when a boolean expression is evaluated by the Python interpreter. They have type bool.

comment
Information in a program that is meant for other programmers

data type
A set of values. The type of a value determines how it can be used in expressions. So far, the types you have seen are integers (int), floating-point numbers (float), and strings (str).

evaluate
To simplify an expression by performing the operations in order to yield a single value.

keyword
A reserved word that is used by the compiler to parse program; you cannot use keywords like if, def, and while as variable names.

object
Also known as a data object (or data value). The fundamental things that programs are designed to manipulate (or that programmers ask to do things for them).

prompt string
Used during interactive input to provide the use with hints as to what type of value to enter.

Introduction to Debugging:

Many a times you think the program you wrote would be doing "abc" only to find it is returning something which is nowhere what you expected.

To take you out of this situation, you need a helping hand which will let you know where you actually went wrong, which is where tool called Debugger comes into picture. As a programmer, one of the first things that you need for serious program development is a debugger.

One of the reasons you will fall in love with Python programming language is because of how easy debugging is. You don't need a full blown IDE to be able to debug your Python application. We will go through the process of debugging a simple Python script using the pdb module from the Python standard library, which comes with most installation of Python.

Programming is a complex process. Since it is done by human beings, errors may often occur. **Programming errors are called bugs** and the **process** of **tracking** them down and **correcting** them is called **debugging**. Some claim that in 1945, a dead moth caused a problem on relay number 70, panel F, of one of the first computers at Harvard, and the term bug has remained in use since. For more about this historic event, see first bug.

As programmers we spend 99% of our time trying to get our program to work. We struggle, we stress, we spend hours deep in frustration trying to get our program to execute correctly. Then when we do get it going we celebrate, hand it in, and move on to the next homework assignment or programming task. But sometimes later when you get to know

something is wrong and the code does not work as you tested anymore and also you didn't add enough debug lines(extra number of prints) it will be a nightmare to go through the code again and then find the problem.

The Mantra for a perfect coding is start small, add enough traces/prints in your code so anytime you run the code again you know what the code is doing.

How to avoid Debugging?

1. **Understand the problem:** You must have a firm grasp over what are you trying to accomplish but not necessarily how to do it. You do not need to understand the entire problem. But you must understand at least a portion of it and what should be the output. This will allow you to test your progress and also keep you aware about what you are coding.
2. **Start small :** As we read above , it is tempting and frustrating to sit down and crank out an entire program at once. Cause when the program does not work, you have a plethora of options of where to start? where to look first? How to figure out what went wrong? . So start with small, make it work fine and then move. This can also be related with **AGILE.**
3. **Keep Improving it:** Once you have a small part of your program working , next step is to identify another small piece of code which can be added and then do the same level of testing , making it efficient.

Another Mantra derived from the earlier mantra is Get something working and keep improving it.

There are some rules to get you thinking and debugging:

1. Everyone is a suspect (Except Python)! It’s common for beginner programmers to blame Python for uneven results, but that should be your last resort. Remember that Python has been used to solve CS1 level problems millions of times by millions of other programmers. So, Python is probably not the problem.
2. Check your assumptions. At this point in your career you are still developing your mental model of how Python does its work. It's natural to think that your code is correct, but with debugging you need to make your code the primary suspect. Even if you think it is right, you should verify that it really is by liberally using print statements to verify that the values of variables really are what you think they should be. You’ll be surprised how often they are not.

Find clues. This is the biggest job of the detective and right now there are two important kinds of clues for you to understand.

Syntax errors:

Similar to other programming languages , Python can **execute** a program if the program is **syntactically correct** or else the process would fail returning an error. **Syntax** refers to the **structure** of a program and the **rules** about the structure.

Unlike reading where you can ignore the syntax errors, Python is not that forgiving. If there is a single syntax error anywhere in your program, Python will display an error message and quit. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. However, as you gain experience, you will make fewer errors and you will also be able to find your errors faster.

For eg:

```
print("Hello World!"
```

```
  File "<ipython-input-3-0219726117bb>", line 1
    print("Hello World!"
                        ^
SyntaxError: unexpected EOF while parsing
```

We have a problem with the formal structure of the program, hence python threw SyntaxError. Python knows where colons/braces/inverted commas are required and can detect when one is missing simply by looking at the code without running it.

Runtime Errors:

This is another type of error which is named as **runtime** because it does not appear until you run the program. These errors are also called as exceptions as they indicate some exceptional handling would be needed to handle them.

```
x=6
y=4
exp=x*y/(4-y)
print(exp)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-2-c56bfb14b659> in <module>
      1 x=6
      2 y=4
----> 3 exp=x*y/(4-y)
      4 print(exp)

ZeroDivisionError: division by zero
```

If you look at the program, it was semantically and the syntax wise correct, but still if the user gives input for y as 4 , the division operand would become 0 and you cannot divide any number by 0.

Python cannot reliably tell if you are trying to divide by 0 until it is executing your program, which might lead to ZeroDivisionError which is a Runtime error.

If an instruction is illegal to perform at that point in the execution, the interpreter will stop with a message describing the exception- ZeroDivisionError.

Semantic Errors:

Another type of error is the Semantic error. If there is a semantic error in your program, it will run successfully in the sense that the computer will not generate any error messages. However, your program will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

This means you did not do a thorough test of what was expected and what code you wrote, meaning the program (it's semantic) is wrong.

Identifying these errors can be tricky as it needs backtrace of your code and finding out what could have made your program to return the output it returned.

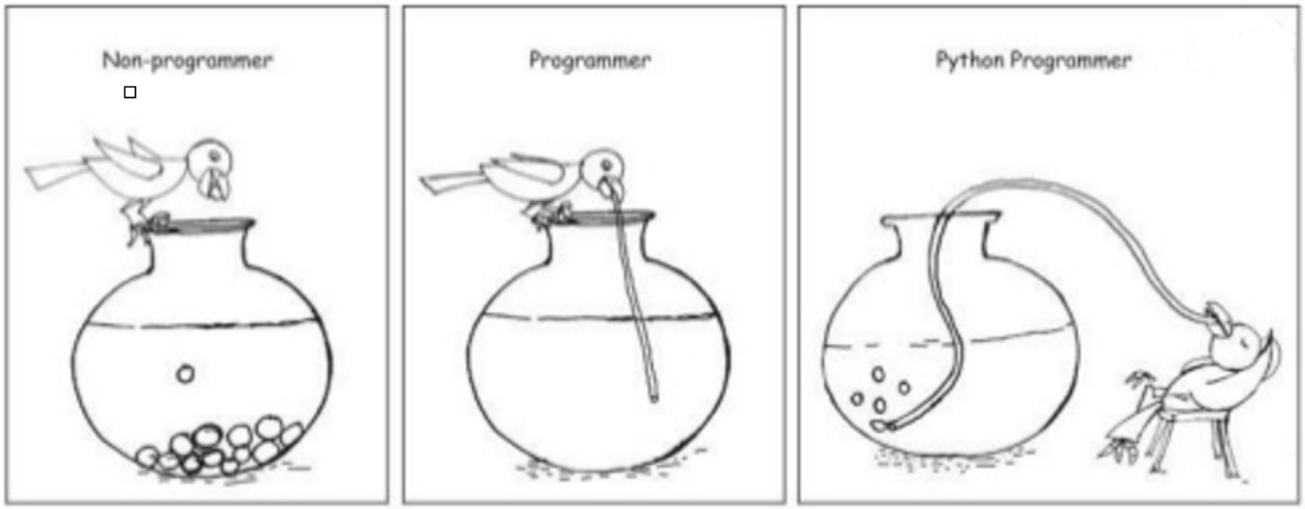
```
x=25
percentage= (25/50)
print('the percentage of your exam is',percentage)
```

```
the percentage of your exam is 0.5
```

For finding percentage you need to multiply it with 100 which was missed hence we have the answer but not what we wanted.

Introduction to Python Modules:

One of the best things about the Python programming language is the ecosystem that surrounds it.



Python provides us the ability to use modules to do everything from sending emails to image processing to complex machine learning. But, even though Python gives us a tremendous amount of power from these modules you still need to know the python basics in order to glue everything together.

Being a C++ developer, this is one of my favorite meme :)

Modules:

Many a times we end up in a situation with a thought **WHAT IF I COULD HAVE A PRE LOADED FUNCTION/CLASS** to make programming easy, this is where Python leaves behind all other languages with its immense amount of community contributing with different libraries/Modules.

Talking about **Module**, it is a file containing python **definitions** and **statements** intended for **use** in other **python programs.** There are many python modules that come with python as part of the standard library. Providing additional functionality through modules allows you to only use the functionality you need when you need it, and it keeps your code cleaner.

People from C++ background must be very well aware about the feature **Namespace.** It is a feature added in C++ , which was not present in C. A namespace is a **declarative region** that provides a **scope** to the **identifiers** (names of the types, function, variables etc) **inside it**. All declarations within those blocks are declared in the named scope.

Snippet from C++:

```
using namespace std;
namespace ns1
 {
      int value()
      { return 5; }
 }
 namespace ns2
{
      const double x = 100;
      double value()
      {  return 2*x; }
}
```

```
int main()
{    // Access value function within ns1
      cout << ns1::value() << '\n';
      // Access value function within ns2
      cout << ns2::value() << '\n';
      // Access variable x directly
      cout << ns2::x << '\n';
      return 0;
 }
```

Output:

```
5
200
100
```

You would have got an idea from above example what namespace is. Functions imported as part of a module live in their own **namespace,** it is simply a space within which all names are distinct from each other. The same name can be reused in different namespaces but 2 objects can't have the same name within a single namespace.

One **example** is the set of street names within a single city. Many cities could have a street called "Main street", "Xyz Square", "North Lane" but it would be very confusing if the same city have the same above names.

Another could be your directory in your filesystem, one folder can have same name but it won't allow you to have another same name file cause that would lead to replication/duplication in data search, while you can have the same named file in some other folder. This helps in maintaining the uniqueness which is the reason why government of all countries has an Identity card with unique numbers.

Importing Modules:

Keyword for importing modules as you might have judged is **import.**
import module_name
remember the name is case sensitive IMPORT/Import does not mean import. when you write import in your program you are indirectly saying "Hey python interpreter, there is some code in other file which i would need, please make its functions and variables in this file." More technically, an import statement causes all the code in another file to be executed.

By convention, all import commands are put at the very top of your file. They can put be elsewhere , but that may lead to some confusions , so it's best to follow the standard convention.

Where do these files are that you import come from? It could be a code file that you wrote yourself, or it code be someone else code that you copied and want to use in your code.

Let's try to understand with an example:
Program name : myfirstprog.py

```
import copiedprog
#code start#
#code end#
```

here **myfirstprog.py** is present in directory **~/Desktop/myfirstcode/**, and **myfirstprog.py** contains a line of code import **copiedprog,** then the python interpreter will look for a file called **copiedprog.py ,** executes its code and make its object bindings available for reference in the rest of the code in **myfirstprog.py**

***KEY POINTS:**

Note that it is **import copiedprog**, not **import copiedprog.py**, but the other file has to be called **myfirstprog.py**

Don't overwrite standard library modules!
For example, if you create a file **random.py** in the same directory where **myfirstprog.py** lives, and then **myfirstprog.py** invokes **import random**, it will **import your file rather than the standard** library module. That's not usually what you want, so be careful about how you name your python files!

random : It is a module which generates pseudo-random numbers with various common distributions.

Syntax for Importing Modules and Functionality:

When you see imported modules in a Python program, there are a few variations that have slightly different consequences.

1. The most common is **import copiedprog**. That imports everything in **copiedprog.py.** To **invoke** a function f1 that is defined in **copiedprog.py**, you would write **copiedprog**.f1().

    - Note that you have to **explicitly mention copiedprog again,** to specify that you want the f1 function from the **copiedprog namespace**.

    - If you just write **f1(), python will look for an f1** that was **defined** in the **current file**, rather than in **copiedprog.py**.

2. You can also give the **imported module an alias** (**a different name**, just for when you use it in your program). For example, after executing **import copiedprog as cp**, you would invoke f1 as **cp.f1().**

- You have now given the morecode module the alias cp. Programmers often do this to make code easier to type.

3. A **third possibility** for importing occurs when you only want to **import SOME of the functionality** from a module, and you want to make those objects be part of the current module's namespace.

- For example, you could write **from copiedprog import f1**. Then you could invoke f1 without referencing **copiedprog** again: f1().

The Random module:

random module is mostly the first module helping us to understand how exactly module feature works in python. We can use random module in a couple of ways.

For example:

1. While playing a lot of games you might come across a game played with dice and we are expecting some random number between 1-6(inclusive) .
2. Playing cards were we need to shuffle and pick any random card.

Let's try to play with random module:

```
import random.py
v=random.random()
print(v)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-11-c1e6935e6bb5> in <module>
----> 1 import random.py
      2 v=random.random()
      3 print(v)

ModuleNotFoundError: No module named 'random.py'; 'random' is not a package
```

Incorrect way:

- As a newbie i had this thought that the random module is actually random.py code so maybe we can use it but as it goes against the convention and is not the right way , we got an error. So correct way is always **import random**

```
import random
v=random.random()
print(v)
```

```
0.948112045650232
```

Correct way:

- The correct way gives you any random number every time you run the code.

```
import random

randomDice = random.randrange(1,7) #randrange is a function defined in random module
print(diceThrow)           #which would return an int, one of 1,2,3,4,5,6
```

```
4
```

Dice throw:

- Everytime you will run the code, the value of randomDice will change

each time. These are random numbers but in a limit which we had defined in the function randrange().

- You might ask why have we kept the boundary as (1-7) while we need output between 1-6, that is because the upper bound is excluded so randrange(1,7) will cover 1-6.

- If you **omit the first parameter** it is assumed to be **0** so randrange(10) will give you numbers from 0-9. **All the values have an equal probability of occurring** (i.e. the results are uniformly distributed).

- random() function returns floating point number in the range [0.0,1.0)-- the square bracket in the left means "closed interval on the left" and round parentheses on right side means "Open interval on the right" Meaning 0.0 is possible but value will never touch 1.0 , always be less than 1.0.

*****KEY POINT:**
It is important to note that random number generators are based on a **deterministic algorithm** — repeatable and predictable.

- Deterministic algorithm is an algorithm which, given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states.
- We use it in mathematical functions.

So they're called **pseudo-random generators** — they are not genuinely random. They start with a **seed value**.

- The seed value is the previous value number generated by the generator. For the first time when there is no previous value, it uses current system time.

Each time you ask for another random number, you'll get one based on the current seed attribute, and the state of the seed (which is one of the attributes of the generator) will be updated.

The catch is that each time you run your program, the seed value is likely to be different meaning that even though the random numbers are being created algorithmically, you will likely get random behavior each time you execute.

- You can also check the current seed value using **random.seed()**

This is the reason , lotteries will never use the random numbers algorithm to list out the numbers cause if anyone ever found out the algorithm behind , they could accurately predict the next value to be generated and would always win the lottery.

Glossary:

namespace
A naming system for making names unique, to avoid duplication and confusion. Within a namespace, no two names can be the same.

random number
A number that is generated in such a way as to exhibit statistical randomness.

random number generator
A function that will provide you with random numbers, usually between 0 and 1.

standard library
A collection of modules that are part of the normal installation of Python.

PART 2