

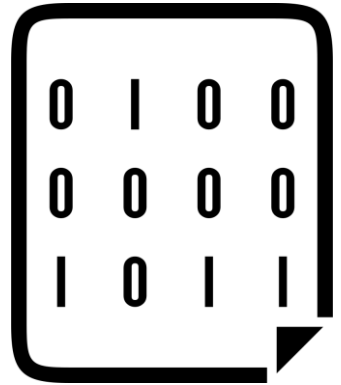
Programming Languages

What is Programming?

- Programming is **writing code** in order to solve a **problem**, using an **algorithm**.
- It is usually written in a **High-Level Programming Language**.
- An HLPL is:
 - Easy and readable
 - Compiled to machine code
- Assembly is a verbal version of machine code.
- Machine code is what a computer knows how to run.

Machine Code - Binary

- This is the only language that the machine understands and is able to run.
- Ones and Zeros (Set/Unset || Lit / Unlit)
- Processor and Operating-System oriented.
- We will not code in this language as it is barely used nowadays.



Assembly

- Readable
- Needs translation into machine code
- Processor oriented

```
MOV ax, 5
MOV bx, 17
ADD ax, bx
```

Low-Level Programming Language

- Machine Code and Assembly are the two main low-level programming languages.
- They are known as “low-level” because they are very close to how different hardware elements of a computer communicate with each other.

High-Level Programming Language (HLPL)

- High-level programming languages are a lot closer to the logic of human communication.
- Easy to use
- More flexible

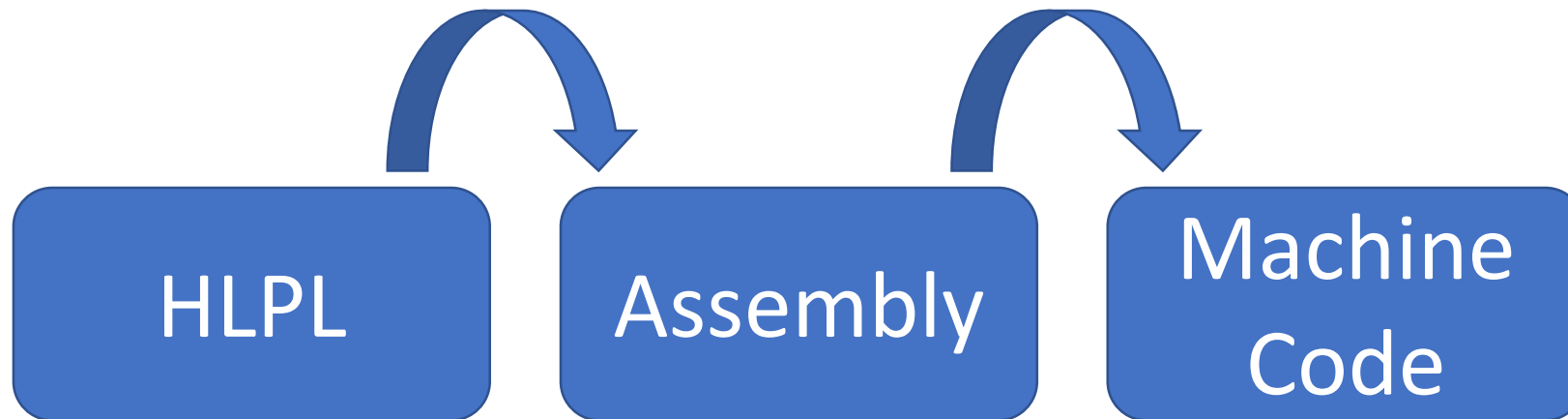
High-Level:

```
x = 17 + 5  
>>> 22
```



Assembly:

```
MOV ax, 5  
MOV bx, 17  
ADD ax, bx
```

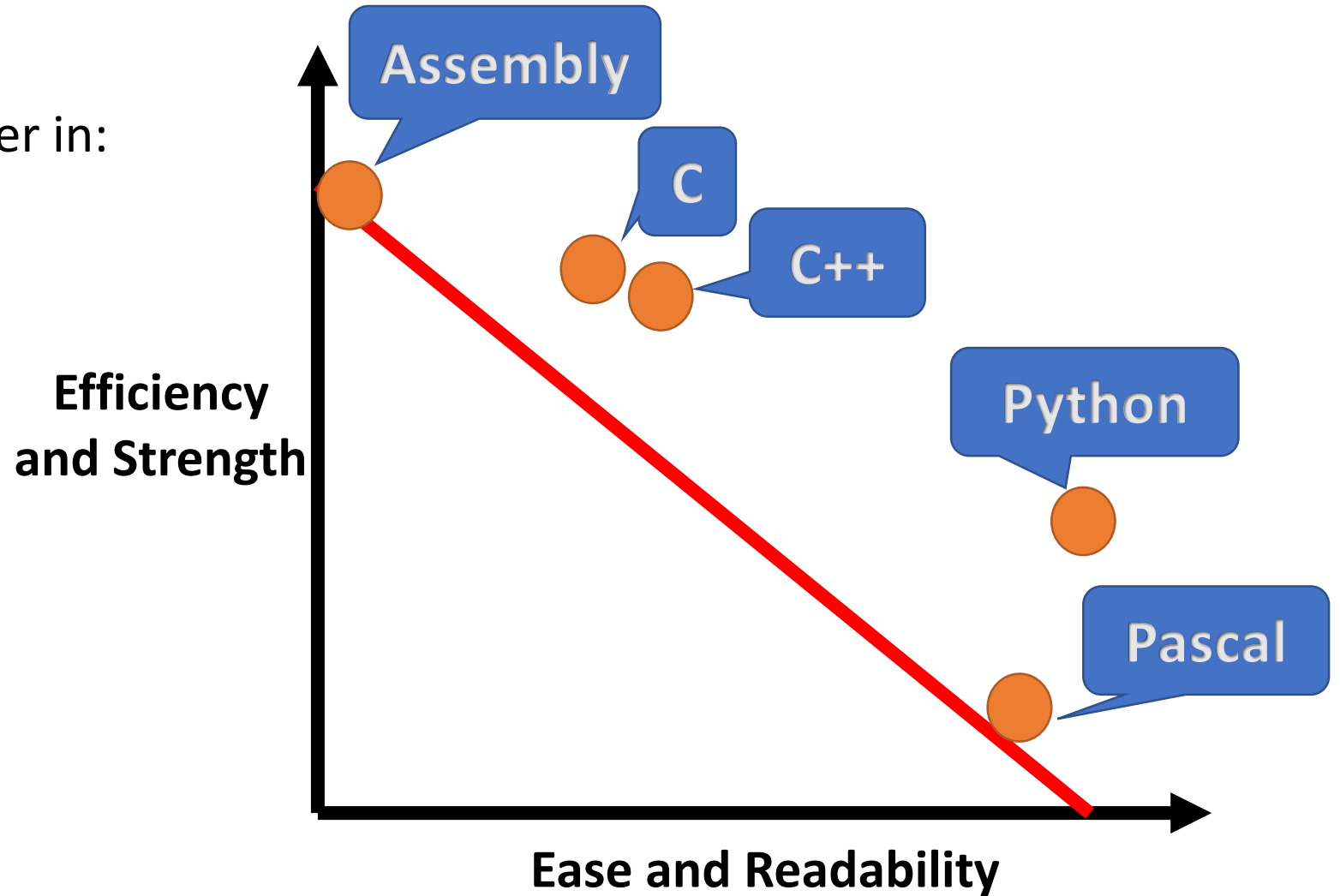


Compiled vs Interpreted Languages

- During compilation, the whole script (code file) is translated from readable language to machine code.
 - This outputs a compiled file, ready for execution. (for example: .exe)
 - At this point, the program still hasn't been run!
 - You can then run this compiled file over and over again.
-
- During interpretation, each line is translated into machine code on its own, and then immediately executed, one line after the other.
 - There is no compiled code.
 - Every time the file is run, each line must be retranslated.

Different Coding Languages

- They differ from each other in:
- Efficiency (Runtime)
- Readability
- Ease of coding



History of Python

- Python 2.0 was first released in 2000. Its latest version, 2.7, was released in 2010.
- Python 3.0 was released in 2008. Its keep updating, with the current latest version of 3.10.
- On January 1, 2020, Python 2.7 was “retired” and is no longer maintained.
- Since Python 2 has been the most popular version for over 15 years, it is still entrenched in the software at certain companies.

Why Python?

- High level
- Interpreted
- Very readable and easy to use
- Often used in the field of cyber security
- Many tools, exploits and POCs are written in python



Programming Language Basics

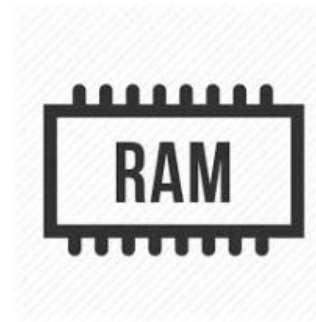
Every programming language contains 4 elements that form the basis of the code's logic:

- **Variables**
- **Conditional Execution**
- **Loops**
- **Functions**

Variables


- Variables are used to store data.
- The data of the variable is stored in the memory.
- The data that is stored in the variable can be changed during runtime.
- In order to store data inside a variable, two parameters are needed:
 - The **Name** of the variable.
 - The **data** that will be stored.

```
x = 12  
y = 'Hello'
```



Variable Modification

$x = 2$  A new variable (x) has been created with the value of 2 .

$x = x + 2$  A new assignment for x has been written.
In order to store the data inside the variable (x), the mathematical expression must be evaluated.

→ Take the variable (x) before $+$ current value (2) and add what is on the right of the upper equals sign (2) to the current value of what is before the $+$ sign.

Variable

Operator

Constant

Assignment Statements

- Is there anything suspicious here?


$$x = 3.9 * x * (1 - x)$$

- In order to use a variable for any kind of operation, it must be defined first.
- In the above statement, we are trying to define x using an expression containing x , however x has not yet been defined.

Numeric Expressions

```
>>> xx = 2
>>> xx = xx + 2
>>> print(xx)
4
>>> yy = 440 * 12
>>> print(yy)
5280
>>> zz = yy / 1000
>>> print(zz)
5.28
```

```
>>> jj = 23
>>> kk = jj % 5
>>> print(kk)
3
>>> print(4 ** 3)
64
```

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Power
%	Remainder

Data Types in Python

Numbers		Strings		
Integer	Float	Single Quotes	Double Quotes	Triple Quotes
Whole Numbers	Floating point numbers	If a character is held in single, double or triple quotes, then we can call it a string.		
Can be either negative or positive	Can be either negative or positive	Values held in single-quotes are equal to values held in double-quotes. However, you CANNOT alternate between them within one string.		Can store single, double-quotes, or multi-line strings.
Int	Float	Str	Str	Str
1, -3, 450, 22	0.33, 1.0, 6.7754	'Text'	"Text"	"""Text"""

Casting

`int()` and `float()` can be used to convert strings and integers.

`str()` can be used to convert a number to a string.

You will get an error if the string does not contain numeric characters.

```
>>> sval = '123'
>>> type(sval)
<type 'str'>
```

```
>>> print(sval + 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int'
```

```
>>> ival = int(sval)
>>> type(ival)
<type 'int'>
>>> print(ival + 1)
124
```

```
>>> nsv = 'hello bob'
>>> niv = int(nsv)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int()
```

The Print() Function

- The `print()` function prints out any value to the screen.

```
name = 'raymond'  
print('Welcome', name)
```

```
>>> Welcome raymond
```

User Input()

- The `input()` function can be used to instruct Python to pause and read data from the user.
- **The `input()` function returns a string.**
- To read a number from the user, the string must be converted to a number using a type conversion function.

```
name = input('Who are you? ')  
print('Welcome', name)
```

```
>>> Who are you? Ray  
>>> Welcome ray
```

```
euf = input('Europe floor? ')  
usf = int(euf) + 1  
print('US floor', usf)
```


String Data Type

- A *String Data Type* is a *sequence of characters*.
- Each character is a *single* letter.

String	'Eddie Rose'										
Char	'E'	'd'	'd'	'd'	'l'	'e'	' '	'R'	'o'	's'	'e'

Defining Strings

- To define a *String* in Python, use either *single quotes* or *double-quotes*.
- The Python interpreter will treat them the *same*!
- This means that we can use either option to define a string; the only difference is how we, as programmers, *decide to use them*.

```
>>> string1 = 'single quotes'
>>> string2 = "double quotes"
```

```
>>> str_w_quote = "Don't you like this?"
>>> str_w_quotes = 'He said "Thank you!" again.'
>>>
```

Multi-line

- There is a third-way to define a string in Python.
- A string can also be defined with triple-quotes! (""" """)
- When triple-quotes are used, Python will ignore everything in between them.

```
>>> string_multiline = """3 double quotes  
creates strings  
spanning
```

```
over multiple lines!!!"""  
>>> string_w_quotes = """  
He hasn't said:  
"Hello" yet! He's late.  
"""  
>>>
```

Length of Strings

- To check the length of a string, the built-in function of *len()* can be used.
- The *len()* function will return the length of a string.

String	<i>Eddie Rose</i>									
Char	'E'	'd'	'd'	'i'	'e'	' '	'R'	'o'	's'	'e'
Length	10									

Indexing

- To find a single letter inside a string, use *square brackets* – `[]` - with the *index* of the wanted letter in them!
- The *index* is the place number of the letter.

String	<i>Cyber Academy</i>												
Char	'C'	'y'	'b'	'e'	'r'	' '	'A'	'c'	'a'	'd'	'e'	'm'	'y'
Index	0	1	2	3	4	5	6	7	8	9	10	11	12

- ***Remember: in Python, we always count starting from 0!***

Indexing

- Strings can be indexed with negative numbers as well!
- In which case, indexing will occur from the end of the string, backward!

String	<i>Cyber Academy</i>												
Char	'C'	'y'	'b'	'e'	'r'	' '	'A'	'c'	'a'	'd'	'e'	'm'	'y'
Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Index	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

String Slicing

- String Slicing is a useful technique to master! Using String Slicing, a new string can be created from an existing string by using the index of that string. If one of the index values is not indicated, Python will use the default, as follows:
 - If the first value is missing, the string will be sliced from the beginning.
 - If the second value is missing, the string will be sliced to the end.

```
my_string[m:a]
```

m → Starting position.

a → Up to, but not including, position.

String Slicing – Third Argument

There is a third argument in String Slicing, which represents Increment/Decrement!

- When a third value is not indicated, Python will use the default → increment by one.

`my_string[m:a:d]`

m → Starting position.

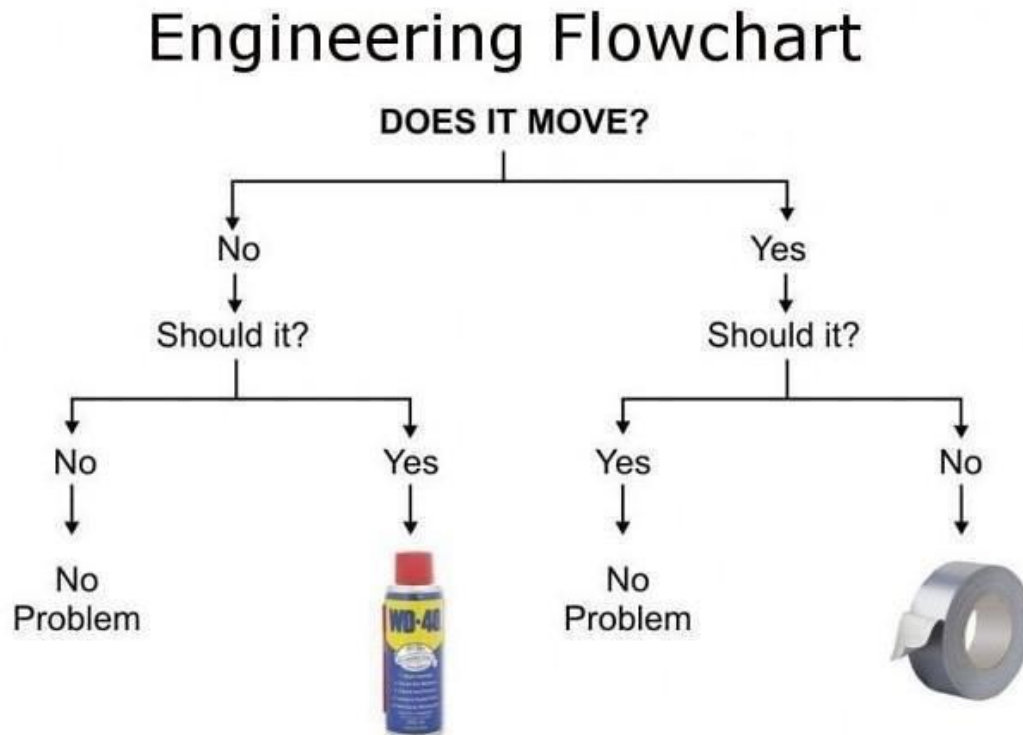
a → Up to but not including position.

d → A stride which indicates how many characters to increment/decrement by.

Conditional Execution

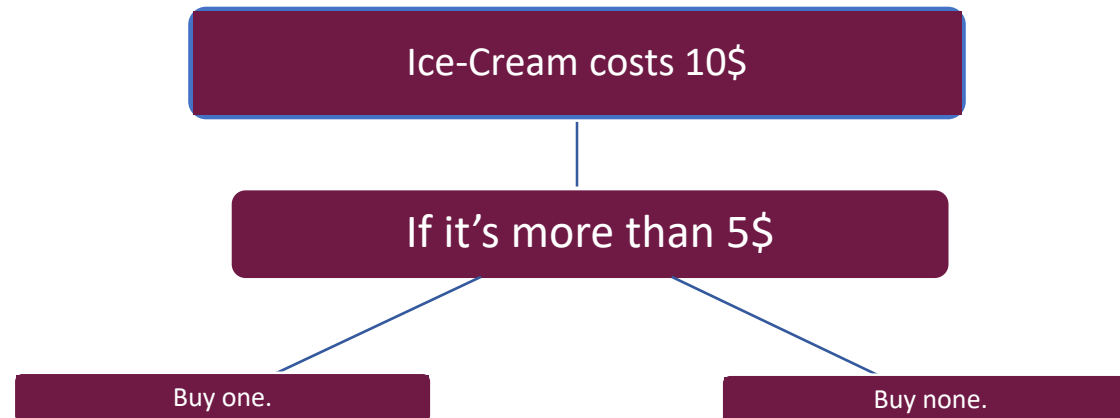
- Conditional execution controls whether a specific block of code will be executed or not.
- The ability to check conditions and change the program's behavior accordingly will almost always be needed to write useful programs!

Example:



IF Statement

- The *if* Statement is used in Python for decision making.
- The *if* Statement will help us evaluate a *Boolean Expression* and determine which code will be executed, respectively.




IF Statement - Syntax

```
if (BOOLEAN EXPRESSION):  
    STATEMENTS
```

- The colon (:) is **significant** and **required**.
- The line after the colon **must** be indented. (4 Spaces)
Python 3 disallows mixing the use of tabs and spaces for indentation.
- All lines indented the same amount after the colon will be executed whenever the *Boolean Expression* is **true**.
- The *Boolean Expression* is called the **Condition**.

The IF Statement

```
x = 5
if x == 5:
    print('Equal to 5')
if x > 4:
    print('Greater than 4')
if x >= 5:
    print('Greater than or equal to 5')
if x < 6:
    print('Less than 6')
if x <= 5:
    print('Less than or equal to 5')
if x != 6:
    print('Not equal to 6')
```



Equal to 5
Greater than 4
Greater than or equal to 5
Less than 6
Less than or equal to 5
Not equal to 6

Else Statement

- The *Else* Statement is used to execute a particular code when our *Boolean Expression* **does not** match our condition.
- Unlike the *If* Statement, which executes code if the *Boolean Expression* returns as **true**, the *Else* Statement can react to a false *Boolean Expression*.

Else Statement - Syntax

```
if (BOOLEAN EXPRESSION):  
    STATEMENTS  
  
else:  
    STATEMENTS
```

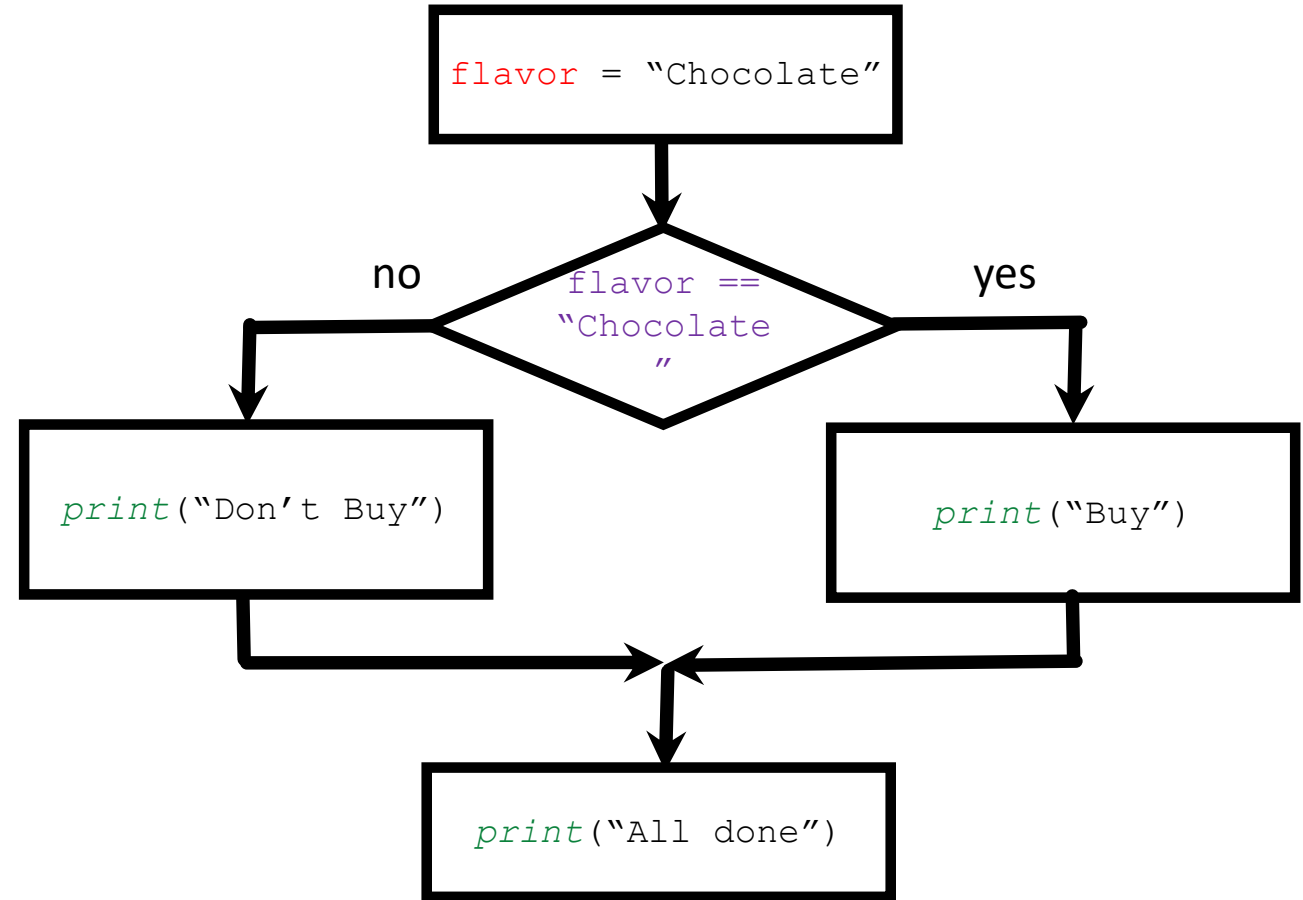
- The colon (:) is **significant** and **required**.
- The line after the colon **must** be indented (4 Spaces).
- All lines indented the same amount after the colon will be executed.

Else Statement

```
flavor = "Chocolate"

if (flavor == "Chocolate"):
    print("Buy")
else:
    print("Don't Buy")

print("All done")
```



Else If Statement

- The *Else If* Statement serves its purpose when we want to execute specific code in cases where our *Boolean Expression* does not match our previous condition, but it might match a new one.
- In some cases, there will be more than two possibilities, and we will need more than one condition.
- Using an *Else If* Statement can be useful in avoiding excessive indentation.
- The keyword '*elif*' is short for '*Else If*'

Lists

- A list is a series of objects or items.
- Lists are defined using square brackets – [].
- They can hold any object within them – even another list!

```
>>> empty_list = []  
>>> clothes_list = ['shirt', 'pants', 'socks']  
>>> all_my_cool_items = ['Lava lamp', 3.141592654,  
2 ** 10, ['a', 'b', 'c']]
```

Indexing – Accessing a Single Item from a List

- Indexing in lists acts just as it does in strings – using square brackets and beginning with an index of 0.

```
>>> all_my_cool_items = ['Lava lamp', 3.141592654,  
2 ** 10, ['a', 'b', 'c']]  
>>> all_my_cool_items[0]  
'Lava lamp'  
>>> all_my_cool_items[-2]  
1024  
>>>
```

0 1 2 3

['Lava lamp' , 3.141592654 , 2**10 , ['a' , 'b' , 'c']]

```
>>> all_my_cool_items [3][1]  
'b'
```

[3][0] [3][1]

Slicing – Accessing a Sub-List of a List

- This, again, is like slicing strings. While string slicing returns a string, accessing a slice of a list returns a list.

```
>>> square_nums = [1, 4, 9, 16, 25, 36]
>>> square_nums[2:5]
[9, 16, 25]
>>> square_nums[::-2]
[36, 16, 4]
>>>
```

Casting

- In earlier lessons, we turned strings into integers, integers into strings, Etc.
- The same can be done with lists, using the `list()` operator!
- In the following example, a **range** object was used, in order to see all elements in the range.
- The same can be done with any object that can be *iterated* over.
- If the object can be treated as a sequence, it can be turned into a list!

```
>>> range(10)
range(0, 10)
>>> list(_)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> letters = 'abcdefg'
>>> list(letters)
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>>
```

List Methods

- There are not many list methods, but every one of them is very useful!

```
>>> dir(list)
['append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
```

Append and Extend

- The **append** method adds a new item to a list:

```
>>> matter_states = []
>>> matter_states.append('solid')
>>> matter_states.append('liquid')
>>> matter_states.append('gas')
>>> matter_states
['solid', 'liquid', 'gas']
```

The **extend** method adds another list to the end of a list:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> a.extend(b)
>>> a
[1, 2, 3, 4, 5, 6]
>>> b
[4, 5, 6]
```

Learning New Methods

- **Q:** How can you teach yourself to use a method you are not familiar with?
- **A:** By using either a question mark - ? – or the help() function!

```
>>> list.remove?
```

```
remove(self, value, /)
```

```
    Remove first occurrence of value.
```

```
    Raises ValueError if the value is not present.
```

Join

- There are two string methods we can discuss now that we understand lists.
- The *join* method (activated on a string) receives a list and returns a string. The new string will be a concatenation of all items in the list, using the original string as a delimiter.

```
>>> word_list = ['These', 'are', 'words', 'in', 'a', 'sentence']
>>> ' '.join(word_list)
'These are words in a sentence'
>>>
>>> letters = list('abcde')
>>> '~#~'.join(letters)
'a ~#~ b ~#~ c ~#~ d ~#~ e'
```


Split

- The ***split*** string method does the exact opposite – it divides a string into a list using a given delimiter.

```
>>> countries = 'Argentina, Brazil, Columbia, Dominican Republic'
>>> countries.split(',')
['Argentina', 'Brazil', 'Columbia', 'Dominican Republic']
```

```
>>> sentence = 'This sentence      has many      spaces'
>>> sentence.split()
['This', 'sentence', 'has', 'many', 'spaces']
```

Loops

- Loops can be helpful whenever we wish to run a block of code multiple times.
- Every loop contains 3 parts:
 - Initialization – This is the initial (starting) state.
Example: $i = 0$ (let's start a loop with a variable 'i' having a value 0)
 - Incrementation/Decrementation – This is how we should increase (increment) or decrease (decrement) our value during each loop.
Example: for each loop, let's add 1 to 'i' – $i += 1$
 - Condition – This is the condition under which the loop should continue. If the condition stops being true, the loop will end.
Example: keep looping until i reaches 10 – $i < 10$

While Loop

- A *while* loop in Python is used to repeat a group of statements, as long as a condition is **true**!
- The '*while*' expression can be thought of as – “As long as..”
- It requires a **condition** and **statements**.
- A *while* loop receives a *Boolean Expression*.
- *As long as* a condition evaluates as *True*, the code block inside the '*while*' loop will execute again and again.
- A *while* loop in Python is created using the '*while*' key-word!

While Loop – Syntax

while Boolean Expression:
Statements

- Starts with '*while*' keyword.
- A *Boolean Expression*.
- A colon. (:))
- The line after the colon **must** be indented. (4 Spaces)
- Indented Statements.

While Loop – Example

```
num = 0
while num < 10:
    print(num)
    num = num + 1
```

- Create a variable and name it “**num**” before-hand!
- Declare a *while* loop using the ‘*while*’ keyword!
- **As long as** the *Boolean Expression* evaluates as true:
 Print out the value of **num**.
 Update the value of **num** by assigning the value of **num + 1** to **num**.
- **Once** the *Boolean Expression* evaluates as *false*, exit the loop!

Infinite Loops

- What is wrong with this loop? This loop will run forever, because n is not incremented!

```
num = 5
while (num > 0):
    print(num)
```

Useless Loops

- What is wrong with this loop? This loop will never run, because the initial condition is never met!

```
num = 0
while (num > 10):
    print(num)
```

For Loop!

- A *for* loop in Python is used to repeat a group of statements, a specified number of times!
- It requires *three* features in order to work.
- → (*Iterating Variable, Statements, Sequence*)
- A *for* loop in Python is declared using the '*for*' key-word!

The Three Features

- In order to use a *for* Loop in Python it must contain **three** features:

Iterating Variable



An indicator
of a starting state

Statement(s)



What to do next

Sequence



Something to loop over

For Loop – Syntax

```
for iterating_var in sequence:  
    Statement(s)
```

- Starts with the '*for*' keyword.
- A *Boolean Expression*.
- A colon. (:))
- The line after the colon **must** be indented. (4 Spaces)
- Indented Statement.

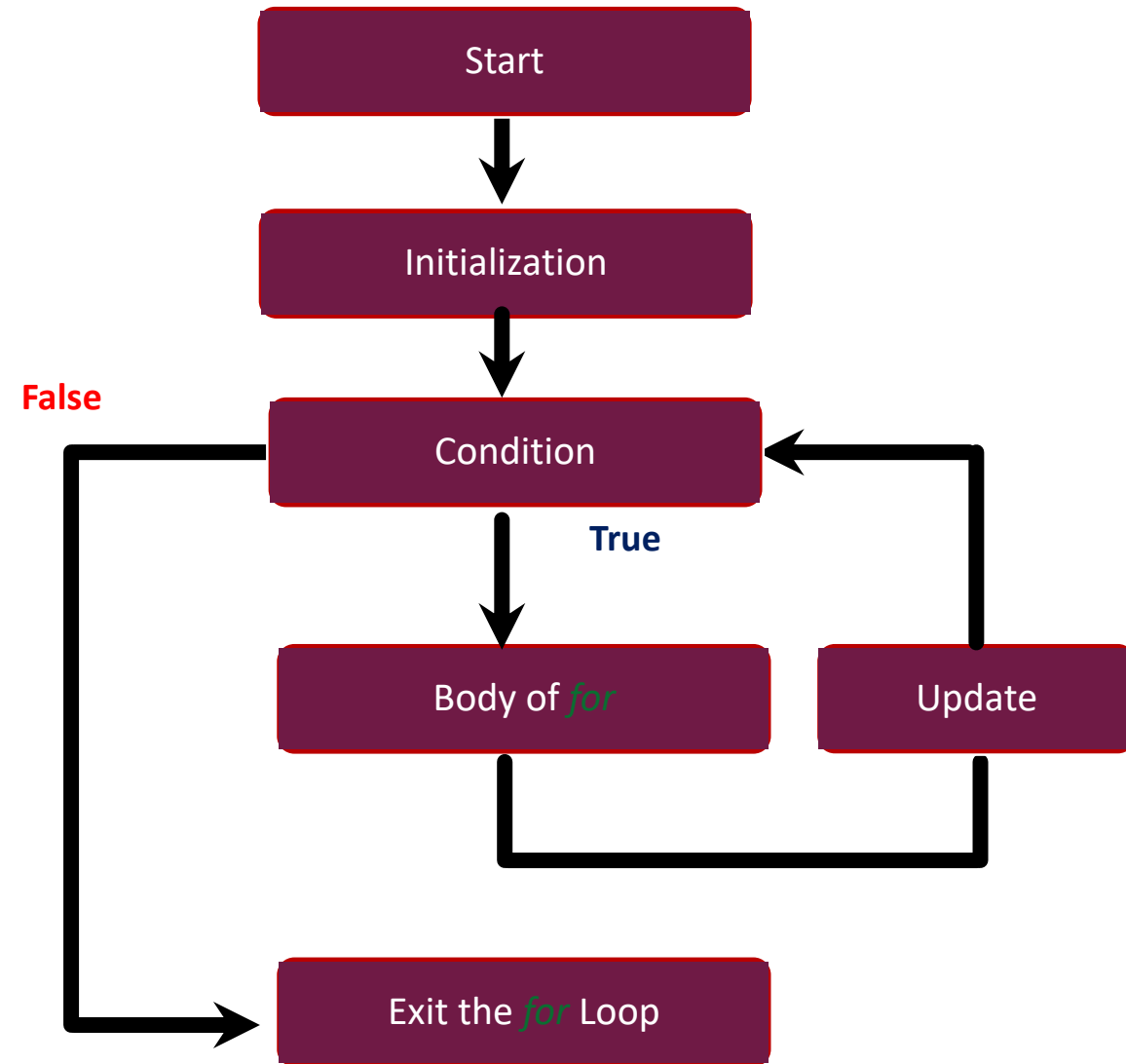
Implementation of the Three Features

```
for letter in 'CYBER':  
    print(number)
```

- *Iterating Variable* → letter
- *Sequence* → 'CYBER'
- *Statement* → *print*(letter)

For Loop – Flow Chart

- *Initialization* is the part where the loop is created.
- *Initialization* is also the part where the **first** value is assigned to the *iterating variable*.
- **Once** the *Boolean Expression* evaluates as *True*, the body of the *for* loop will be entered and the *statements will be executed*!
- **Once** the first iteration is complete, the *iterating variable will be updated*, and the *Boolean Expression* will be checked once again!
- **Once** the *Boolean Expression* evaluates as *False*, the *for* loop will be exited!



range() Function – List View

- A *range()* is similar to a *list*!
- Example - Calling out *range*(10):

range(0, 10)

Will return the numbers 0,1,2,3,4,5,6,7,8,9 (it **will not** return 10).

- We can use the *list()* function to cast a range *sequence* into a *list*!

```
list(range(10))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

For Loop – Example

```
for number in range(10) :  
    print(number)
```

- Declare a for loop using the *for* keyword!
- Create an *Iterating Variable* “**number**”!
- Create a *sequence* of digits from 0 to 9, using *range(10)*.
- During the *first iteration*, the *first* value in the *sequence* will be assigned to *number* (0).
- *For each iteration* of the *sequence*, the statement will be executed: print out the value stored inside *number*.
- Finally, the value stored inside *number* will be set to the next value stored inside the *sequence*!

Looping Over Numbers - Demonstration

```
for number in range(10):  
    print(number)
```

```
for number in [0,1,2,3,4,5,6,7,8,9]:  
    print(number)
```



Both loops will produce the same output:

0
1
2
3
4
5
6
7
8
9

For Loop example

```
for n in range(1,101):  
    if n%2 == 0 :  
        print(n , 'is even')  
    else:  
        print (n , 'is odd')
```

- The expression `n%2` checks the remainder of the division of `n` by 2.
- If the remainder of the division is equal to 0, the number is even, and if not, the number is odd.
- This loop will run on all the numbers between 1 and 100 (pay attention to the range that appears in the command), and checks if the number is even or not.

For Loop Variable Names

- *Iterating Variables* i, j, and k are the conventional variable names used for loops.
- That being said, if the iterations have meaning, a more indicative name should be used, to improve readability:

```
for celebrity_name in ['The Rock', 'Bruno Mars', 'Cristiano Ronaldo']:  
    print(celebrity_name)
```


Basic Function Creating

- Steps:

1. Use the ***def*** keyword
2. Function name
3. Brackets
4. Colon
5. Indented function body

- This defines (*but does not execute*) the function.

```
def say_hi():  
    print('Hi!')
```

Arguments & Parameters

- A function can be added to arguments as inputs. These can also be called *parameters*.
- For example, *input* receives the argument *prompt*, which is the question that the user will be asked.
- The function *print* receives inputs to print to the screen.
- These arguments are written inside the parentheses, when the function is called.
- **Dictionary Corner:**
- ***Argument*** – the input given to the function when *calling/executing* the function.
- ***Parameter*** – the input given to the function when *defining* the function.
- These are essentially the same but used in different contexts.

Single Parameter Example

Execution:

```
def say_hi(language):  
    if language == 'es':  
        print('Hola')  
    elif language == 'fr':  
        print('Bonjour')  
    elif language == 'ar':  
        print('Salaam')  
    elif language == 'en':  
        print('Hello')  
    else:  
        print('Unknown language...')
```

```
In [12]: say_hi('ar')  
Salaam
```

```
In [13]: say_hi('en')  
Hello
```

```
In [14]: say_hi('zz')  
Unknown language...
```

Will the Below Function *'return'* Anything?

- This function, much like the function *print*, prints to the screen but doesn't ***return*** anything.
- In order to return a value, we should use the built-in keyword ***return***.

```
def say_hi():  
    return 'Hi!'
```

- The return statement ends the function execution and “sends back” the result of the function.

Using *return*

- Again, once the code reaches ***return***, the function execution ends, and the value is given back to whatever called the function.

```
def say_hi(language):  
    if language == 'es':  
        return 'Hola'  
    elif language == 'fr':  
        return 'Bonjour'  
    elif language == 'ar':  
        return 'Salaam'  
    elif language == 'en':  
        return 'Hello'
```

```
greeting = say_hi('ar') + ', Hugh!'  
print(greeting)
```

```
C:\Python>python functions2.py  
Salaam, Hugh!
```