1 Introduction

1.1 What is programming? Why learn it?

- A process is a series of subtasks that transforms some **inputs** into some **outputs** to accomplish a high-level task.
- Process is kind of vague - "How do I do this thing?"
- An algorithm (in a general sense) is a series of precise steps to accomplish a computation.
- Algorithm is super precise - "What is the list of steps (with cases, subcases etc) to do this thing?"
- A program is an algorithm written in a programming language for input to a computer.
- **As a programmer, your job will be to convert** a

> high-level task -> process (vague idea) -> algorithm (precise ideas) -> program (in Python/Java/your favourite PL)

So what kind of tasks can you write programs for? If you can describe it precisely enough - then you can code it.

Let's look at a few examples.

The MSc admission process at XYZ College is currently being handled by the Student Affairs department. But it is too much workload - so it is decided that the college will have to hire 4 new employees to purely handle admissions. But Rama, the head of SA, suggests that why don't we automate this process to reduce the workload instead? Can this be done? Let's take a look.

Task : Conduct MSc admissions to XYZ College.
Inputs: Student Applications
Outputs: Results

There is already a process in place, let's look at it.

- Students fill up a form with questions about biodata, previous education etc.
- Students mail the form to Student Affairs department.
- Student Affairs (SA) filters out the ineligible candidates based on minimum grades, compatibility of previous degreee with applied program etc.
- Sorts the remaining applications based on Department.
- Sends the bundles to each department.
- The department professors evaluate each application and send back a list of application numbers which are accepted and another which are rejected.
- SA crosschecks to see that no student has been missed out.
- SA sends mails to each of the students informing them of the result.
- This is just step 1... after this students will send back their acceptances and based on that, there will be second round of counselling etc.

This is a task with a very **precise** process already existing for it. Rama is right - it is a perfect candidate for automation.

The ABC school always faces trouble during exams, due to high volume of exam papers to correct. Typically, they hire about 4 teachers part-time each year during exam season. Inspired by Rama, her brother Ravi who works at the ABC school suggests that instead of hiring external teachers why don't they hire a software developer to write code to automatically correct the papers. Is this possible? Let's take a look. Let's start with one subject - English.

Task : Automatically assign marks out of 10 for an English essay.
Inputs : .txt file with English essay.
Outputs: Marks between 0 to 10.

To figure this out, Ravi asked Raktim who has been teaching English for 10 years how he corrects his papers. This is what Raktim says

- Cut 1/2 marks for every spelling mistake. But cut a maximum of 3 marks for spelling mistakes.
- Cut 1/2 marks for every grammatical mistakes. But cut a maximum of 3 marks for grammar mistakes. But don't cut more than 5 marks in total for spelling and grammar.
- Assign 4 marks based on how well the student has covered the topic. Take into account whether they have given pros/cons if applicable.
- Assign 2 marks if the student has managed to add some personal view or narrative to the topic.
- Assign 4 marks for structuring and fluency of the essay.

Raktim has a really good process - it is very precise for an English teacher. But it is still very difficult to convert to a program. Consider the following issues-

- How do you assess how well the student has covered the topic?
- How do you judge the structure and fluency?
- How do you know if a personal view has been reflected?

All these things are possible to do for Raktim because of either human being's innate ability to understand language, or Raktim's experience as an English teacher. But since it can't be put in **exact** instructions - this is not a good candidate for converting to a program.

But in 10 years time, Ravi might be right - as AI and ML is trying to come up with precise answers to these vague problem!

Now let's take a simple example of a task and try to write a program for it.

Situation:

- You work for a very old fashioned Data analysis company that doesn't like to use computers.
- Your job is to solve **2 linear equations in 2 variables**, which you do with pen and paper.
- Your boss says he wants to promote you to solving **3 linear equations in 3 variables** (yay!).
- But you also have to handle your old job, because they are very cheap and won't hire anyone to do your old job.
- This year, they have finally bought a few computers.
- You have an idea, how about you write a Python program to do your old job?

First what is the task?

Task : Solve 2 linear equations in 2 variables.
Inputs : 2 equations in x and y.
Outputs : Values/answers for x and y.

Okay, good. Now for the most important part - what is a process to solve it?

- If you're confused about how to get the process, let's start with a small example - and see how you would solve it.

Example Input:

```
x+2y=10
2x+y=8
```

- Can we solve using elimination? Let's try

  ```
  First eliminate x by matching coefficients

  2*(x+2y)-(2x+y)=2*10-8
  => 3y=12
  => y=4

  Now substiting y in eqn 1,

  x+8=10
  => x=2
  ```

Excellent! You have solved the equation successfully. \

Now can we make this more general ?

General Input:

```
ax+by=c
ux+vy=w
```

Now a,b,c,u,v,w can be any numbers! Can we generalize the previous logic?

```
```
First eliminate x by matching coefficients

u*(ax+by) - a*(ux+vy) = u*c -a*w
=> (ub-av)*y = uc-aw
=> y= (uc-aw)/(ub-av)

Now substiting y in eqn 1,

ax+by=c
=> x = c/a - b/a *y
=> x = c/a - b/a * (uc-aw)/(ub-av)

```
```

Now we have successfully got a general **algorithm** for solving the task!

- Just one note : Do you need the entire equations as input?

Program

- Now let's convert that into code

In [1]:

```
def solve_equations(a,b,c,u,v,w):
    # solve system of linear equations
    # ax + by = c
    # ux + vy = w

    ## First eliminate x, how?

    # Multiply eqn 1 by "u", and eqn 2 by "a" and subtract them
    # Then y = (c*u - w*a)/(b-v)
    y=(c*u-w*a)/(b*u-v*a)

    # Now get x from eqn 1

    x = (c - b*y)/a

    # return the values

    return x,y
```

executed in 3ms, finished 12:38:54 2020-05-11

In [2]:

```
solve_equations(1,2,10,2,1,8)
```

executed in 9ms, finished 12:38:54 2020-05-11

Out[2]:

```
(2.0, 4.0)
```

In [3]:

```
solve_equations(5,2,22,2,1,8)
```

executed in 4ms, finished 12:38:54 2020-05-11

Out[3]:

```
(6.0, -4.0)
```

This code is super easy to understand, its just like how you would write the calculations for your boss!

But now you can run it for any inputs your boss gives you! Also you have used the following Pythonic concepts!

- Variables
- Expressions
- Comments
- Functions

1.2 Brief Intro to Python

Quickly here are a few points about Python

- Python is designed to read like English. "Maximum readability" is the mantra to follow while writing Python.
- Python has few keywords and is simple to learn.
- Python is an interpreted language. This means every line is executed one by one when you run the code. Which means that there are no "compile time errors".
- You can use the Python interpreter for an interactive programming experience.
- Python has a very large ecosystem of Machine Learning libraries making it a go-to for ML and DS practitioners.
- Python is also a popular choice for writing backends for web-applications, especially with the Django framework.

If this tutorial is not enough, you might want to browse the following tutorials

- THE Python Tutorial (https://docs.python.org/3/tutorial/)
- TutorialsPoint (https://www.tutorialspoint.com/python/index.htm)

1.2.0.1 Installing Python

- If you are on Linux or MacOS, you already have python installed.
- Open up a terminal and type `python` to start the interpreter.

- You can install Miniconda (https://docs.conda.io/en/latest/miniconda.html) to manage environments and packages.
- Alternatively, you can also install Anaconda (https://docs.anaconda.com/anaconda/install/) which is Miniconda + a bunch of other software and packages. It also has a GUI interface.
- Anaconda comes fully equipped with all the popular data science packages and Jupyter Notebook, the software we are using right now.
- For Miniconda you have to install everything by yourself, but that's easy.
- I highly recommend **Miniconda** - this way you know exactly what you're putting into your computer.

What is Jupyter?

- Jupyter is just a way to mix text and python code. Just imagine that everytime you run a code block -> you just send that code to the interpreter running behind this website -> exactly like if you typed it out.

If you have installed Miniconda, you just need the following commands

First create your environment

```
conda create --name eict python=3.6
```

Then, enter your environment

```
conda activate eict
```

You can install packages like this

```
conda install jupyter
```

- If you have installed Anaconda, you can open Jupyter Notebook from the Anacoda Navigator.
- You can manage environments and packages from the navigator as well as by using conda with the above commands.

> [Installing Miniconda on Windows Youtube (https://www.youtube.com/watch?v=q04RsYo0f1w)](https://www.youtube.com/watch?v=q04RsYo0f1w)

2 Datatypes and Operators

- What are the simplest types of data in Python?
- How do you represent them?
- What can you do with them?

2.1 Numbers

- Two basic types of numbers in Python you should be concerned about for now
  - Integers
  - Floats

In [4]:

```
# integer : no decimal
print(53)

# float : decimal
print(2.003)
```

executed in 3ms, finished 12:39:01 2020-05-11

```
53
2.003
```

- The `print` **function** is a convenient way to print to the **console** (or actually stdout).
- Don't worry about what functions are now. Just remember it prints out whatever is in between the brackets

2.1.0.1 Operators

- Arithmetic Operators : + - / * % ** //
  - Takes two numbers and returns a number
  - 3 / 2 ---> 1.5
  - If one of the numbers is a float, the result is a float.
  - If both are integers, the result is an integer.
    - Exception : /
- Comparison Operators : > < <= >= == !=
  - Return a boolean - True, False

Notes :

- Bitwise operators
- Operator precedence : BODMAS rule or just use brackets!

2.2 Booleans

In [5]:

```
# booleans
print(True)
print(False)
```

executed in 3ms, finished 12:39:02 2020-05-11

```
True
False
```

Boolean operators : **and** , **or** , **not**

```
True and False
True or False
not True
```

2.3 Strings

In [6]:

```
print("this is a string")
print('you can also use single quotes')
```

executed in 3ms, finished 12:39:03 2020-05-11

```
this is a string
you can also use single quotes
```

- Indexing and Slicing

  ```
  "My name is"[1]
  "My name is"[-1]
  "My name is"[1:5]
  "My name is"[1:]
  "My name is"[1:-5]
  ```

- Concatenation

  ```
  "My name " + " is " +" Aneesh"
  ```

- Repitition

  ```
  "Repeating"*3
  ```

- Membership operators

  ```
  "M" in "America"
  "M" not in "America"
  ```

    - Returns a boolean - True or False
    - In this case second one is true, because "M" is NOT in "America". Remember that in coding letters are just symbols, so M and m are different.

- Escaping quotes in strings

```
"A single quote ' inside a double quote"

'Or a double quote " inside a single quote"

"Or just use \\ to escape \""
```

- Newlines in strings

```
"A newline is written like this - \n"
```

Instructor Note : Move to Python interpreter and demonstrate

- All the binary operators for numbers, booleans and strings
- Nested expressions with paranthesis
- Implicit typecasting for numbers
- Mixing comparison operators and boolean operators

2.4 Converting between datatypes

- Python provides a bunch of very simple functions, that does the job of type conversion for you.

- Between float and int

In [7]:

```
# Converting float to integer, takes the floor of the float
print(int(22.9))
print(float(33))
```

executed in 3ms, finished 12:39:04 2020-05-11

```
22
33.0
```

- Boolean to number

In [8]:

```
print(int(True))
print(int(False))
```

executed in 4ms, finished 12:39:05 2020-05-11

```
1
0
```

- Numbers and Booleans to Stringa

In [9]:

```
print(str(13))
print(str(49.99))
print(str(True))
```

executed in 3ms, finished 12:39:05 2020-05-11

```
13
49.99
True
```

- String to numbers

In [10]:

```
print(int("22"))
print(float("33.3333"))
```

executed in 3ms, finished 12:39:05 2020-05-11

```
22
33.3333
```

- Other datatypes to boolean
  - Everything is true except for some special values - 0, 0.0, "", None

In [11]:

```
print(bool(2))
print(bool(-3))
print(bool(43.1))
print(bool("hello"))
print(bool("\n"))
```

executed in 4ms, finished 12:39:06 2020-05-11

```
True
True
True
True
True
```

In [12]:

```
print(bool(0))
print(bool(0.0))
print(bool(""))
print(bool(None))
```

executed in 4ms, finished 12:39:06 2020-05-11

```
False
False
False
False
```

3 Variables

- A variable is a name for a piece of data.
- It is not that piece of data itself, it is just the name.
- To create a variable, you just assign some data to the name.

In [13]:

```
just_a_name=0
```

executed in 2ms, finished 12:39:07 2020-05-11

So now this name now refers to an integer, but you can replace it with any sort of data.

In [14]:

```
just_a_name="This is a string"
print(just_a_name)
just_a_name=22.9
print(just_a_name)
just_a_name=True
print(just_a_name)
```

executed in 3ms, finished 12:39:07 2020-05-11

```
This is a string
22.9
True
```

When you put something in the box using the syntax

```
x = "my value"
```

this is called an **assignment**.

3.0.0.1 What happens if the RHS contains a variable?

```
x=y
```

- Well then, now x refers to the same data as y.

> See this [blog post (https://medium.com/broken-window/many-names-one-memory-address-122f78734cb6)](https://medium.com/broken-window/many-names-one-memory-address-122f78734cb6) for a great explanations on assignments in python and how they relate to memory addresses, immutability and "interning".

3.0.0.2 Note : Assignment + Operator short syntax

In python code you will often see code like this

```
x+=2
x*=44+22
```

This is just a short syntax for when the RHS is an expression with the first operand as x. It is exactly the same as

```
x=x+2
x=x*(44+22)
```

- People who know C might recognize that these are there in C as well.
- They might be wondering if the ++ operator is there in python too.
- The answer is NO.
- Reason : The BDFL hates the ++ operator :P

4 Conditional Statements

Core idea : Based on the value of a variable, you want to do different actions/computations.

4.1 The basic if statement

- Your boss tells you to design personalized greeting messages for users coming to your website.
- You don't have time, so you come up with a simple rule to fake personalization depending on her/his name.

In [15]:

```python
student_name="Ravina"
if student_name[0] in "ABCDEFGHIJKLMN":
    print("Hello "+student_name+" welcome to XYZ corp!")
else:
    print("Namaste "+student_name+"! XYZ corp is very happy to see you!")
```

executed in 4ms, finished 12:39:09 2020-05-11

```
Namaste Ravina! XYZ corp is very happy to see you!
```

The generic syntax of an if statement is

```
if condition :
    x=22
    y=x*z
    ...
    #body
    ...
```

- Indentation of 4 spaces
- Condition is any expression that evaluates to a boolean
  - Otherwise it will be converted to a boolean automatically, but beware!

4.1.0.1 A note on indentation, whitespace and code blocks in Python

- if statements are the first time some of you might be encountering code blocks.
- Code blocks are a chunk of code, that is guaranteed to run from top to bottom sequentially.
- Python uses indentation(spaces) to differentiate code blocks.
- Code blocks are used inside conditional statements and loops only in Python.

In [16]:

```
x=20
```

executed in 2ms, finished 12:39:10 2020-05-11

In [17]:

```
if x==20:
    print("This is inside the block and so will only print if x==20")
print("This is outside, so will always print")
```

executed in 3ms, finished 12:39:10 2020-05-11

```
This is inside the block and so will only print if x==20
This is outside, so will always print
```

In [18]:

```
this_is=20
    will_this="work"
```

executed in 3ms, finished 12:39:10 2020-05-11

```
IndentationError: unexpected indent (<ipython-input-18-5e82ebcdc697
>, line 2)
```

In [19]:

```
this_is=20
will_this="work"
```

executed in 2ms, finished 12:39:10 2020-05-11

In [20]:

```
if 21<33:
x=21
```

executed in 4ms, finished 12:39:10 2020-05-11

```
IndentationError: expected an indented block (<ipython-input-20-c1b
345444c33>, line 2)
```

In [21]:

```
if 21<33:
    x=21
```

executed in 2ms, finished 12:39:11 2020-05-11

4.1.0.2 Differences with code blocks in languages like C

- In C, whitespace doesn't matter.
  - In Python, whitespace at the beginning of a line are very important.
- In C, Code blocks are marked by { curly brackets }.
  - In Python, whitespace at the beginning of the line mark a code block.
- You can create interior code blocks anywhere.
  - In Python, an interior level of code can only be created inside a conditional statements body or a loop's body.
- In C, variables defined within an interior code block are deleted/released once you leave that block.
  - In Python, variables defined in an inner block are still available to the outer block.
  - Read about scope (https://en.wikipedia.org/wiki/Scope_(computer_science)) in programming languages.

In [22]:

```
if 21 < 25 :
    a_totally_new_variable="secret value"
print(a_totally_new_variable)
```

executed in 3ms, finished 12:39:12 2020-05-11

```
secret value
```

4.2 else statements and nesting

In [23]:

```
x=22
```

executed in 2ms, finished 12:39:12 2020-05-11

In [24]:

```
if x < 100 :
    print("from -inf to 99")
else :
    print("from 100 to +inf")
```

executed in 3ms, finished 12:39:12 2020-05-11

```
from -inf to 99
```

- The else statement takes care of the exclusionary case. Everything that is not part of the if, goes to the else.
- Let's look at something slightly more complicated.

In [25]:

```
if x <=0 or x>100:
    print("Not a number from 1 to 100")
else:
    print("A number from 1 to 100")
```

executed in 3ms, finished 12:39:13 2020-05-11

```
A number from 1 to 100
```

Tip : Practice calculating the **opposite** of a condition.

- Opposite of >? Opposite of ==?
- Opposite of **x and y**? Opposite of x?

- Suppose you want to break up the **if case** or the **else case**.
- You can put another if...else statement inside the **if code block** or the **else code block.**

In [26]:

```
if x <=0 or x>100:
    if x <=0:
        print("Negative number!")
    else:
        print("Positive number from 101 to inf")
else:
    print("A number from 1 to 100")
```

executed in 4ms, finished 12:39:13 2020-05-11

```
A number from 1 to 100
```

In [27]:

```
if x <=0 or x>100:
    print("Not a number from 1 to 100")
else:
    if x <50:
        print("Number from 1 to 50")
    else:
        print("Number from 51 to 100")
```

executed in 3ms, finished 12:39:13 2020-05-11

```
Number from 1 to 50
```

Tip :

- Practice drawing simple branching diagrams, to explain how to divide a problem into multiple subcases and sub sub cases. Visualizing it really helps - you can visualize it however you want, branching diagrams are just one example.
- Even for experienced programmers, if...else statements are the place where they make the most mistakes!

- Conditions are confusing, so always try to write it in a readable way that is very simple to understand.
- For example, we can rewrite the first code by switching the if...else conditions to make it more readable.

In [28]:

```
if x > 0 and  x<=100:
    print("A number from 1 to 100")
else:
    print("Not a number from 1 to 100")
```

executed in 3ms, finished 12:39:14 2020-05-11

```
A number from 1 to 100
```

4.3 elif statements

In [29]:

```python
if x <=0 or x>100:
    print("Not a number from 1 to 100")
else:
    if x <50:
        print("Number from 1 to 50")
    else:
        if x <=75 :
            print("Number from 51 to 75")
        else:
            print("Number from 76 to 100")
```

executed in 4ms, finished 12:39:14 2020-05-11

```
Number from 1 to 50
```

Now this is getting confusing... We can simplify this using the **elif** statement.

In [30]:

```python
if x <=0 or x>100:
    print("Not a number from 1 to 100")
elif x <50:
    print("Number from 1 to 50")
elif x <=75 :
    print("Number from 51 to 75")
else:
    print("Number from 76 to 100")
```

executed in 4ms, finished 12:39:15 2020-05-11

```
Number from 1 to 50
```

- The above two blocks of code are exactly the same.

The logic behind if...elif...else blocks is like this.

1. Look at the **if statement**
   A. If it matches, execute interior code. Then **SKIP ALL the elif and else blocks**.
   B. If it doesn't match go to next elif.
2. Look at the **elif statement**
   A. If it matches, execute interior code. Then SKIP ALL the **remaining** elif and else blocks**.
   B. If it doesn't match go to next elif.

```
....
....
....
```

3. You have reached the **else statement**.
   A. This can only happpen if none of the above if or elif statements matched!

- Thus, if...elif...else is a prioritized list of cases.
- Therefore **order** matters.
- The below two codes have different meanings

In [31]:

```
if x <100 :
    print("Less than 100")
elif x <200 :
    print("Between 100-200")
```

executed in 3ms, finished 12:39:16 2020-05-11

```
Less than 100
```

In [32]:

```
if x <200 :
    print("Less than 200")
elif x <100 :
    print("Unreachable code")
```

executed in 3ms, finished 12:39:16 2020-05-11

```
Less than 200
```

4.4 pass statement

- The `pass` statement is another special statement.
- It does ... nothing.
- So why do we need it?
- Although `pass` statement can be put anywhere...

In [33]:

```
pass
```

executed in 2ms, finished 12:39:16 2020-05-11

- It's most (only?) valid use case is in an if....else statement.
- This is when you want to write down a case very clearly, but you don't actually need to do anything for it. Or you want to specifically ignore it.

In [34]:

```
if x < 0 :
    print("Negative")
elif x <100:
    pass
else:
    print("Large positive number")
```

executed in 3ms, finished 12:39:17 2020-05-11

5 Loops

5.1 The basic while loop

In [35]:

```
i=0
while i < 10:
    print(i)
    i+=2
```

executed in 4ms, finished 12:39:18 2020-05-11

```
0
2
4
6
8
```

The syntax for the basic while loop is

```
while condition :
    x+=1
    y=x+42
    ...
    ...
    # body
    ...
print("outside the loop")
```

- Uses indentation to mark inside code block just like if...else
- Conditions are expressions which must evaluate to a boolean i.e. True or False statements

5.1.1 Example 1 : Calculating n factorial

n! = n * (n-1) * (n-2) .... * 2 * 1

For example,

5! = 5 * 4 * 3 * 2 * 1

In [36]:

```
n=5
product=1
current=1
while current <= n:
    product*=current
    print(product)
    current+=1
```

executed in 4ms, finished 12:39:18 2020-05-11

```
1
2
6
24
120
```

Notice

- `current` is the loop variable. It's what is changing as the loop changes.
- More importantly it is what causes the loop to **terminate**.

What happens if we forget to add the line `current+=1` ?

In [ ]:

```
n=5
product=1
current=1
while current <= n:
    product*=current
    print(product)
```

executed in 14.5s, finished 12:39:34 2020-05-11

BEWARE OF INFINITE LOOPS!!!

- Infinite loops happen because the condition at the top of the while loop is never reached.
- Even experienced programmers regularly make mistakes that lead to infinite loops.
- Code is all about conditions, since if...else and while are enough to write practically any sort of computation.
- So if the conditions are wrong, you can run into issues like infinite loops.

5.1.2 Example 2 : Printing cool ascii art X

In [38]:

```
size=10
i=0
while i <size:
    string=""
    j=0
    while j < size:
        if j==i:
            string=string+"\\"
        elif j==size-i-1:
            string=string+"/"
        else:
            string=string+"-"
        j+=1
    print(string)
    i+=1
```

executed in 6ms, finished 12:39:37 2020-05-11

```
\--------/
-\------/-
--\----/--
---\--/---
----\/----
----/\----
---/--\---
--/----\--
-/------\-
/--------\
```

Notice :

- You can nest loops
- You can mix loops with
- You might notice two variables `i` and `j`.
- Both are loop variables for each loop.
- Can you use the same variable for both loops?
    - In principle you can, and in some cases it might even make the code more readable.
    - But that's a dangerous idea!
    - Make KISS - Keep It Simple Silly - your mantra.
    - If you keep your coding style as simple and easy to understand as possible, you can tackle more and more complex problems.

5.2 A brief intro to Lists

- So far you have seen simple datatypes such as numbers, strings and booleans.
- And, actually data comes in these "basic" forms only (well, more or less).
- But what if you have lots of these small data chunks? How do you arrange them?

- One of the simplest ways to arrange data is in a list. Take a real life list for example

My courses for this year

- Artificial Intelligence
- Computer Vision
- Data Mining
- Randomized Algorithms
- Data Structures
- English
- Lists are an **ordered** collection of objects.

In Python you define lists like this

In [39]:

```python
# a list with some elements
my_courses = ["Artificial Intelligence","Computer Vision",
              "Data Mining","Randomized Algorithms",
              "Data Structures","English"]

# btw, str() works to get a string representation of most things in Python!
print("A list of strings : " + str(my_courses))
```

executed in 3ms, finished 12:39:38 2020-05-11

```
A list of strings : ['Artificial Intelligence', 'Computer Vision', 'Da
ta Mining', 'Randomized Algorithms', 'Data Structures', 'English']
```

Side Note :

- By the way, do you notice I split the definition into multiple lines for neatness?
- In general, Python does not allow a statement to continue to the next line by itself.
- You have to use "" to split a statement like this

```
x = 22 + \
    33 + \
    44
```

- But exceptions to this rule is for list/dictionary/tuple/set definitions and function/constructor/method calls and definitions.

In [40]:

```
# an empty list
empty_list = []
print("An empty list : "+str(empty_list))
```

executed in 3ms, finished 12:39:39 2020-05-11

```
An empty list : []
```

In [41]:

```
# lists can have mixed values
mixed_values_list = [1, True, 100.0, "HEllo"]
print("A mixed list : "+str(mixed_values_list))
```

executed in 4ms, finished 12:39:39 2020-05-11

```
A mixed list : [1, True, 100.0, 'HEllo']
```

In [42]:

```
# lists can even contain lists! Actually lists can contain any object!(More later)
list_of_lists=[22,33,[100,22],[11,444,0]]
print("A mixed list with list elements : "+str(list_of_lists))
```

executed in 4ms, finished 12:39:39 2020-05-11

```
A mixed list with list elements : [22, 33, [100, 22], [11, 444, 0]]
```

Reading values from the list

You can do it by indexing and slicing, very similar to strings.

In [43]:

```
print(my_courses[0])
print(my_courses[-1])
print(my_courses[1:])
print(my_courses[3:5])
print(my_courses[-4:-3])
print(my_courses[-1:])
```

executed in 5ms, finished 12:39:40 2020-05-11

```
Artificial Intelligence
English
['Computer Vision', 'Data Mining', 'Randomized Algorithms', 'Data Stru
ctures', 'English']
['Randomized Algorithms', 'Data Structures']
['Data Mining']
['English']
```

If you try to index outside the list, you will get an IndexError.

In [44]:

```
mylist=["only","three","elements"]
print(mylist)
mylist[3]
```

executed in 8ms, finished 12:39:40 2020-05-11

```
['only', 'three', 'elements']
```

```
IndexError: list index out of range ▸
```

Assigning new values to the list

You can do this by indexing the list like below

In [45]:

```
print(mixed_values_list)
```

executed in 3ms, finished 12:39:41 2020-05-11

```
[1, True, 100.0, 'HEllo']
```

In [46]:

```
mixed_values_list[1]=42
print(mixed_values_list)
```

executed in 3ms, finished 12:39:41 2020-05-11

```
[1, 42, 100.0, 'HEllo']
```

Note :

- Adding new elements to a list (and making it longer) is also possible. It will be discussed in a subsequent section.
- Similarly for deleting elements.

5.3 The basic for loop

- One of the main reasons to write a loop is to work with lists!
- Think about it, lists have a lot of usually similar elements, and you probably want to do the same operation for each of them.
- Thus Python has a special syntax to work with lists and list-like objects called the **for loop**.

In [47]:

```
numbers=[2,4,3,5]
for num in numbers:
    square=num*num
    print(square)
```

executed in 3ms, finished 12:39:42 2020-05-11

```
4
16
9
25
```

Thus the generic syntax for for loops is

```
for element in iterable:
    # body of the loop
    print(element)
```

- An "iterable" is something that you can "iterate" over - basically it has a bunch of smaller elements, which it can give to you one by one.
- A list is an iterable.
- A string is also an iterable!

In [48]:

```
for letter in "Strings are iterable!":
    print("||     " + letter + "     ||")
```

executed in 5ms, finished 12:39:42 2020-05-11

```
||     S     ||
||     t     ||
||     r     ||
||     i     ||
||     n     ||
||     g     ||
||     s     ||
||           ||
||     a     ||
||     r     ||
||     e     ||
||           ||
||     i     ||
||     t     ||
||     e     ||
||     r     ||
||     a     ||
||     b     ||
||     l     ||
||     e     ||
||     !     ||
```

But if you try with a number, obviously a number is not iterable...

In [49]:

```
for something in 22:
    lets="try this"
```

executed in 7ms, finished 12:39:43 2020-05-11

```
TypeError: 'int' object is not iterable
```

5.4 break and continue statements

- Think of `break` and `continue` as commands to use within a loop.
- `break` tells the loop to stop.
- `continue` tells the loop, okay just go to the next iteration, no need to finish this one.
- `break` and `continue` can only be used inside a loop.

In [50]:

```
break
```

executed in 4ms, finished 12:39:44 2020-05-11

```
SyntaxError: 'break' outside loop (<ipython-input-50-6aaf1f276005>,
line 4)
```

In [51]:

```
animals=["cat","cat","cat","dog,""cat","cat"]
only_cats=True
for animal in animals:
    if animal!="cat":
        only_cats=False
        break
print(only_cats)
```

executed in 4ms, finished 12:39:44 2020-05-11

```
False
```

In [52]:

```
numbers=[1,5,22,67,87,33,2,34]
find_this=67
i=0
for num in numbers:
    if num==find_this:
        print(str(find_this)+" found at index "+str(i))
        break
    i+=1
```

executed in 5ms, finished 12:39:44 2020-05-11

```
67 found at index 3
```

In [53]:

```
numbers=[1,5,22,67,87,33,2,34]
print("Printing only odd numbers in list")
for num in numbers:
    if num%2==0:
        continue
    print(num)
```

executed in 5ms, finished 12:39:45 2020-05-11

```
Printing only odd numbers in list
1
5
67
87
33
```

- **You don't usually need break and continue** statements. But they are VERY useful.
- They are just another tool for you.
- Use them if it makes it easier for you to translate your ideas into code.

For example, the above example can be rewritten without `continue`.

In [54]:

```
numbers=[1,5,22,67,87,33,2,34]
print("Printing only odd numbers in list")
for num in numbers:
    if num%2!=0:
        print(num)
```

executed in 4ms, finished 12:39:45 2020-05-11

```
Printing only odd numbers in list
1
5
67
87
33
```

This is probably actually better this way :)

6 Functions

- For code to be reusable, you have to "pack it up".
- Think of it like a link to a webpage or a note - you don't want to keep writing the same thing over and over again - so you write it once and refer to it afterwards.

In [55]:

```
# You define a function like this
def print_fancy_name(firstname,lastname):
    print("-"*15)
    print(firstname)
    print(lastname)
    print("*"*15)
```

executed in 3ms, finished 12:39:46 2020-05-11

- The `def` keyword is specially used to define functions.
- These lines of code won't actually run, they'll just add this definition of the function to the interpreter's memory.
- You can't use a function before you define it!

In [56]:

```
my_greeting("Ram")

def my_greeting(name):
    print("Wassup "+name+"!")
```

executed in 6ms, finished 12:39:46 2020-05-11

```
NameError: name 'my_greeting' is not defined ▸
```

6.1 Arguments

- Notice that this function has inputs - these are called arguments
- You have to call the function with actual values for the arguments to run it.

In [57]:

```
# You call a function like this
print_fancy_name("Hritik","Roshan")
```

executed in 3ms, finished 12:39:47 2020-05-11

```
---------------
Hritik
Roshan
***************
```

In [58]:

```
# You can reuse this for someone else now
print_fancy_name("Sunidhi","Chauhan")
```

executed in 3ms, finished 12:39:47 2020-05-11

```
---------------
Sunidhi
Chauhan
***************
```

- As you can see, you can have multiple arguments to the function or even no arguments.

In [59]:

```
def say_hello():
    print("hello")

say_hello()
```

executed in 4ms, finished 12:39:48 2020-05-11

```
hello
```

- Arguments can be specified by position or name

In [60]:

```
def family_tree(first_name,last_name,father_name,mother_name):
    print("-"*15)
    print(first_name)
    print(last_name)
    print("*"*15)
    print("Child of "+mother_name+" and "+father_name)
```

executed in 3ms, finished 12:39:48 2020-05-11

In [61]:

```
# positional
family_tree("Ravina","Sharma","Ram Sharma","Kusum Sharma")
```

executed in 3ms, finished 12:39:48 2020-05-11

```
---------------
Ravina
Sharma
***************
Child of Kusum Sharma and Ram Sharma
```

In [62]:

```
# keyword -- order doesn't matter
family_tree(father_name="Ram Sharma",last_name="Sharma",
            first_name="Ravina",mother_name="Kusum Sharma")
```

executed in 4ms, finished 12:39:49 2020-05-11

```
---------------
Ravina
Sharma
***************
Child of Kusum Sharma and Ram Sharma
```

In [63]:

```
# mixed positional + keyword
# Obviously positional has to come first
family_tree("Ravina","Sharma",mother_name="Kusum Sharma",
            father_name="Ram Sharma")
```

executed in 4ms, finished 12:39:49 2020-05-11

```
---------------
Ravina
Sharma
***************
Child of Kusum Sharma and Ram Sharma
```

In [64]:

```
# this will fail
family_tree(mother_name="Kusum Sharma","Ravina","Sharma",
            father_name="Ram Sharma")
```

executed in 3ms, finished 12:39:49 2020-05-11

```
SyntaxError: positional argument follows keyword argument (<ipython
-input-64-a01a694b9521>, line 2)
```

6.2 Return Values

Functions can also return values, for example suppose we want a function to calculate the square of the difference of two numbers

In [65]:

```
def square_difference(x,y):
    diff=x-y
    return diff*diff
```

executed in 2ms, finished 12:39:50 2020-05-11

In [66]:

```
print(square_difference(5,2))
```

executed in 3ms, finished 12:39:50 2020-05-11

```
9
```

So what is the return value of `print_fancy_name` ?

In [67]:

```
print(print_fancy_name("Priyanka","Chopra"))
```

executed in 3ms, finished 12:39:51 2020-05-11

```
---------------
Priyanka
Chopra
***************
None
```

- It is a special datatype called **None**. It signifies nothing. It is used whenever you want to say that there is nothing here.

In [68]:

```
x=None
print(x)
print(bool(x))
```

executed in 3ms, finished 12:39:52 2020-05-11

```
None
False
```

- If there is no return statement, like `print_fancy_name` None is automatically returned.
- Its the same case for when due to the some reason, the return statement is not executed.
- Also, if there is an empty return statement.

In [69]:

```
def give_toffee_to_kids_only(age):
    if age<13:
        return "toffee"

print(give_toffee_to_kids_only(5))
print(give_toffee_to_kids_only(25))
```

executed in 3ms, finished 12:39:52 2020-05-11

```
toffee
None
```

In [70]:

```
def give_toffee_to_kids_only_2(age):
    if age<13:
        return "toffee"
    return

print(give_toffee_to_kids_only_2(5))
print(give_toffee_to_kids_only_2(25))
```

executed in 4ms, finished 12:39:53 2020-05-11

```
toffee
None
```

- Unlike languages like C,Java,C++, Python allows you to return multiple values

In [71]:

```
def return_initials(first_name,middle_name,last_name):
    return first_name[0],middle_name[0],last_name[0]

print(return_initials("John","Winston","Lennon"))
first,middle,last=return_initials("James","Paul","McCartney")
print(first+" "+middle+" "+last)
```

executed in 4ms, finished 12:39:53 2020-05-11

```
('J', 'W', 'L')
J P M
```

Note : Why the brackets around the first output?

- That's because Python packs them into tuples! More on that later.

6.3 Example 1 : Return to control flow

Let's look at a slightly longer example for functions now.

In [72]:

```
def is_right_angled_triangle(a,b,c):
    if a*a + b*b == c*c:
        return True
    elif a*a + c*c == b*b:
        return True
    elif b*b + c*c == a*a:
        return True
    else:
        return False
```

executed in 4ms, finished 12:39:54 2020-05-11

It should be noted, that once the interpreter sees the return statement, it will exit the function - even if there is more code afterwards.

In [73]:

```
def print_hello(name):
    print("Hello "+name)
    return
    print("Now I can write whatever I want here")
    print("It will not execute")
```

executed in 3ms, finished 12:39:55 2020-05-11

In [74]:

```
print_hello("Ram")
```

executed in 3ms, finished 12:39:55 2020-05-11

```
Hello Ram
```

Exploiting this fact we can rewrite the above function by removing all the `elif` s

In [75]:

```
def is_right_angled_triangle(a,b,c):
    if a*a + b*b == c*c:
        return True
    if a*a + c*c == b*b:
        return True
    if b*b + c*c == a*a:
        return True
    return False
```

executed in 4ms, finished 12:39:56 2020-05-11

- Remember there are many ways to write the same logic as code, just as there are many ways to communicate the same idea with words!
- Always try to refactor your code for maximum readability.

6.4 Example 2 : Recursion

- Recursion is a common design pattern in functions.
- Recursion is when a function calls itself.
- Usually, you start with a "big" problem, and if you can break it up into a smaller problem where the same logic has to be applied you can call the same function on it.
- Finally, you should end up with a small enough problem - whose answer you know - called the base case.

The classic example of recursion is - calculating n!

We know,

```
n! = n * (n-1) * ... * 1
   = n * (n-1)!
```

Now you can see that we can represent n! in terms of (n-1)!.
So maybe we can write a recursive function for it!

In [76]:

```
def factorial(n):
    print(n)
    return n * factorial(n-1)
```

executed in 3ms, finished 12:39:57 2020-05-11

In [ ]:

```
factorial(5)
```

executed in 1.92s, finished 12:40:00 2020-05-11

- Oops what just happened?
- The danger of calling a function **recursively** is infinite recursion!
- Infinite recursion usually happens when the base case does not apply correctly, in this case we totally forgot to write the base case.
- Imagine it like falling through the floors of a building (like a cartoon!)... If someone forget's the floor - you'll fall forever!

In [78]:

```
# lets try again
def factorial(n):
    print(n)
    if n==1:
        return 1
    return n * factorial(n-1)
```

executed in 3ms, finished 12:40:03 2020-05-11

In [79]:

```
factorial(5)
```

executed in 5ms, finished 12:40:03 2020-05-11

```
5
4
3
2
1
```

Out[79]:

```
120
```

- This time it works!

Let's look at another example

In [80]:

```
def check_palindrome(string):
    if len(string)<=1:
        return True

    if string[0]==string[-1]:
        return check_palindrome(string[1:-1])
    else:
        return False
```

executed in 4ms, finished 12:40:04 2020-05-11

In [81]:

```
print(check_palindrome("abcba"))
print(check_palindrome("abba"))
print(check_palindrome("abcbas"))
```

executed in 4ms, finished 12:40:04 2020-05-11

```
True
True
False
```

6.5 Built-in functions

Python provides many built in functions that are always available (as opposed to? Read about imports).

The print function

In [82]:

```
print("A single string")
print("Still "+"a single"+"string is passed")
print("Many","strings","are","passed")
print("Some may not even be strings :",1,True,print)
```

executed in 5ms, finished 12:40:05 2020-05-11

```
A single string
Still a singlestring is passed
Many strings are passed
Some may not even be strings : 1 True <built-in function print>
```

The input function (for taking console input)

In [83]:

```
year=input("Enter your birth year\n")
print("You are now ",2020-int(year),"years old!")
```

executed in 4.83s, finished 12:40:10 2020-05-11

```
Enter your birth year
1996
You are now  24 years old!
```

In [84]:

```
# REPL - Read Eval Print Loop
# A common design pattern for interactive command line applications.
# When you run the Python interpreter - that's an example of a REPL!
# In fact the command line itself is a REPL!
while True:
    year=input("Enter your birth year or x to exit\n")
    # A good example of break
    if year=="x":
        break
    print("You are now ",2020-int(year),"years old!")
```

executed in 6.26s, finished 12:40:17 2020-05-11

```
Enter your birth year or x to exit
1992
You are now  28 years old!
Enter your birth year or x to exit
2001
You are now  19 years old!
Enter your birth year or x to exit
x
```

Some math related functions

In [85]:

```
print(abs(-22.44))
print(abs(22.1))
```

executed in 3ms, finished 12:40:17 2020-05-11

```
22.44
22.1
```

In [86]:

```
print(max([1,2,4,5]))
print(max(6,3))
print(min([1,2,4,5]))
print(min(6,3))
```

executed in 4ms, finished 12:40:21 2020-05-11

```
5
6
1
3
```

In [87]:

```
print(pow(9,1/2))
print(sum([2,5,8]))
```

executed in 3ms, finished 12:40:21 2020-05-11

```
3.0
15
```

The len() function

In [88]:

```
print(len([1,2,3,4]))
print(len("How long?"))
```

executed in 3ms, finished 12:40:22 2020-05-11

```
4
9
```

The id() function

In [89]:

```
x="abcd"
y=x
print(id(x))
print(id(y))
print(id(x)==id(y))
```

executed in 4ms, finished 12:40:23 2020-05-11

```
4398318344
4398318344
True
```

In [90]:

```
help()
```

executed in 9.26s, finished 12:40:32 2020-05-11

```
Welcome to Python 3.6's help utility!

If this is your first time using Python, you should definitely check
out
the tutorial on the Internet at https://docs.python.org/3.6/tutoria
l/. (https://docs.python.org/3.6/tutorial/.)

Enter the name of any module, keyword, or topic to get help on writin
g
Python programs and using Python modules.  To quit this help utility
 and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, typ
e
"modules", "keywords", "symbols", or "topics".  Each module also come
s
with a one-line summary of what it does; to list the modules whose na
```

See this link (https://docs.python.org/3/library/functions.html) for the full list of built-in functions.

7 Classes, Objects, Attributes & Methods (15 min)

- You saw that functions were a way to "pack up" some piece of logic.
- What if you want to "pack up" both logic and data?
- We won't be writing classes but we will be using them so it's important that you get a basic grasp of **classes, objects, constructors, attributes and methods**

In [113]:

```python
# you define a class just like you define a function
class Family:

    # the constructor of the class
    # the constructor is where you do setup for the object
    # usually this means to define attributes of the object
    def __init__(self,father_name,mother_name,children_names,
                 children_ages,home_address):
        # self refers to the current object
        # you can define new attributes inside the object just like you
        # define variables in Python
        self.father_name=father_name
        self.mother_name=mother_name
        self.children_names=children_names
        self.children_ages=children_ages
        self.home_address=home_address

    # this is a method - notice the first argument is always self
    # self is an implicit argument which you don't have to pass
    # when calling the method.
    def print_diwali_invitation(self):
        print("*"*25)
        print("Hello, please come for our Diwali celebration at "+
              self.home_address)
        print("from")
        print(self.mother_name+" and "+self.father_name)
        print("and our wonderful children")
        namestring=""
        for childname in self.children_names:
            namestring+=childname+","
        namestring=namestring[:-1]
        print(namestring)
        print("*"*25)

    def return_youngest_child(self):
        min_index=0
        min_age=1000
        for i in range(0,len(self.children_names)):
            if self.children_ages[i]<min_age:
                min_age=self.children_ages[i]
                min_index=i
        return self.children_names[min_index], min_age

    def change_address(self, new_address):
        self.home_address=new_address
```

executed in 7ms, finished 13:27:02 2020-05-11

- A class is a template or blueprint to pack some data and logic.
- Think of it as a blueprint for a box of "subvariables" called "attributes" and "subfunctions" called methods.
- Generally methods will operate on the attributes - that's why we pack them together.

In [92]:

```
sharmas=Family("Ram Sharma","Kusum Sharma",
               ["Ravi Sharma","Rani Sharma","Komal Sharma"],
               [21,15,9],"Geetanjali Residency, Kondapur, Hyderabad")
```

executed in 3ms, finished 12:40:37 2020-05-11

- You instantiate a class using the following syntax

  `x=ClassName(arg1,arg2,arg3...)`

- What you get is an "object" which is a particular instance of a class.
- That is, you took the blueprint and made a real box and filled it with real data.

In [100]:

```
vijs=Family("Prateek Vij","Kareena Vij",
            ["Lucky Vij","Ravi Vij"],[13,5],
            "Geetanjali Residency, Kondapur, Hyderabad")
```

executed in 3ms, finished 12:44:08 2020-05-11

- You can create another box - i.e. another object - it has the same blueprint (same attributes, methods) but different values for data.
- The functions (logic) however remains the same.

In [101]:

```
# You can access the attributes/data in an object using "." notation
print(sharmas.children_names)
```

executed in 2ms, finished 12:46:10 2020-05-11

```
['Ravi Sharma', 'Rani Sharma', 'Komal Sharma']
```

In [102]:

```
# You can access the methods/logic in an object also using "." notation
sharmas.print_diwali_invitation()
```

executed in 4ms, finished 12:46:53 2020-05-11

```
*************************
Hello, please come for our Diwali celebration at Geetanjali Residency,
Kondapur, Hyderabad
from
Kusum Sharma and Ram Sharma
and our wonderful children
Ravi Sharma,Rani Sharma,Komal Sharma
*************************
```

In [103]:

```python
# We need classes and objects so we can have different values of attributes,
# and then run the same logic on them.
sharmas.print_diwali_invitation()
vijs.print_diwali_invitation()
```

executed in 4ms, finished 12:48:40 2020-05-11

```
*************************
Hello, please come for our Diwali celebration at Geetanjali Residency,
Kondapur, Hyderabad
from
Kusum Sharma and Ram Sharma
and our wonderful children
Ravi Sharma,Rani Sharma,Komal Sharma
*************************
*************************
Hello, please come for our Diwali celebration at Geetanjali Residency,
Kondapur, Hyderabad
from
Kareena Vij and Prateek Vij
and our wonderful children
Lucky Vij,Ravi Vij
*************************
```

- Let's look at how we created an object again.
- This looks like a function call.
- Actually you are calling the "constructor" of the class
  - This is the "subfunction" `__init__` in the definition of the class.
  - So you have to pass all the arguments of the constructor just like a function.
  - Except the first argument - `self` . You don't have to pass `self` , Python does that automatically for you.
- The constructor is where you do setup for the object.
  - Usually this means just creating the attributes of the object
  - You can create a new attribute for an object anytime by writing `objectvariable.new_attr="some value"` .
  - Within the constructor of course the variable for the object is `self` so you write `self.new_attr="some value"` .
  - Technically, you don't need to setup attributes in the constructor - you can do it later also. Like you can have a method like `family.setup_attributes("..."....)` .
  - But obviously that doesn't make sense, when the constructor is meant for doing setup work.

In [106]:

```python
# notice : same arguments as __init__
# notice : self was not passed
vijs=Family("Prateek Vij","Kareena Vij",
            ["Lucky Vij","Ravi Vij"],[13,5],
            "Geetanjali Residency, Kondapur, Hyderabad")
```

executed in 3ms, finished 13:03:34 2020-05-11

In [107]:

```python
# If you use the dir function, you can see all methods and attributes
dir(vijs)
```

executed in 3ms, finished 13:03:43 2020-05-11

```
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'change_address',
 'children_ages',
 'children_names',
 'father_name',
 'home_address',
```

In [110]:

```python
# just like in the constructor we wrote self.father_name=father_name
# we can add attributes here too
# as long as we have a variable containing/pointing to the object
vijs.has_dog=True
print(vijs.has_dog)
# will "has_dog" appear when we run dir(vijs) now?
```

executed in 3ms, finished 13:06:08 2020-05-11

```
True
```

In [112]:

```python
# generally you'll have this sort of pattern to change attributes of an object
sharmas.change_address("Lumbini Enclave, Whitefields, Hyderabad")
print(sharmas.home_address)
# although we can even do it like this
sharmas.home_address="Geetanjali Residency, Kondapur, Hyderabad"
print(sharmas.home_address)
```

executed in 3ms, finished 13:07:26 2020-05-11

```
Lumbini Enclave, Whitefields, Hyderabad
Geetanjali Residency, Kondapur, Hyderabad
```

- Everything that was said about positional arguments, keyword arguments, return values etc for **functions** applies to methods and the constructor as well.

- Now you understand that a class is just a way **to pack some data with functions**.
- So, now we can tell you a secret -- In Python everything is an object! We will deal with this in detail in the next section!

8 End of Module 1 !

Congratulations, now you know a little bit about everything!

- Datatypes (Numbers,Strings,Booleans)
- Operators
- Variables
- Assignments
- Conditional Statements (if...else)
- Loops (while and for)
- Data Structures (Lists)
- Functions
- Classes, Objects, Attributes and Methods

But there were many things that are missing because they require a mix of all these concepts. So we will cover them in the next section!

- Instructor Note : Now would be a good time to quickly revise Part 1.