Table of contents

Python For ML

First of all `python` is a very easy langauge to learn and use. And for machine learning we can start by learing the very basics of programming in python.

Because there are `many libraries` that you will learning thoughout this article series that need very little python knowledge but a lot of `mathematical` theory.

So, in this article we will go throguh the essential `python` concepts that you will need to know to `get started` with `machine learning`.

- LinkedIn
- YouTube
- gtihub
- Gmail
- discord

Installation

We need `conda`, which is a package manager that comes with `Anaconda` distribution. You can download it from here or you can install `miniconda` which is a smaller version of `Anaconda` that only includes the `conda` package manager. You can download it from here.

`Conda` has `python` pre-installed, so you don't need to install it separately. But a good practice is to create a new environment but let's just stick to the default environment for now.

> Full installation is in the ML github repository

Setting Up

Now that we have `conda` installed, we can start coding but how do we code? It's python so we can just directly code in a python file right?

Technically speaking, we can code in a python file but there is a better way to code for ML and AI. That is by using `Jupyter Notebooks`.

Now, what is that, you ask?

> Jupyter Notebooks are a web-based interface for creating and executing code, markdown, and text in a single document. They are often used for data analysis, scientific computing, and machine learning.

This is exactly like a python file but we can do a lot more than just write python code in it.

So, if you installed `miniconda` you should be able to start coding in a `jupyter notebook` right now.

Just go to the terminal and activate the base environment and run the following command:

```
conda activate # This will activate the base environment
Jupyter notebook # This will start a jupyter notebook
```

> This should automatically open a new tab in your default web browser with the jupyter notebook interface and you should see your directory structure in the interface.

> The directory will be the exactly same as the directory you are in in the terminal.

For some of you the `jupyter notebook` may not open automatically and show an error message. In that case, `base` environment doesn't have `jupyter notebook` installed. You can install it by running the following command:

```
conda install jupyter
```

This should install `jupyter notebook` and all its dependencies in the `base` environment.

And you should see a new tab in your web browser with the jupyter notebook interface when you run the command.

BUUUUUUT. I'll not be using web browser for this series. I'll be using `vscode` for this series.

So, how do we setup `vscode` to work with `jupyter notebook`?

Setting up VS Code

First we need the jupyter extension.

Now make a new directory/folder in your desktop or any other place of you choice and open it in vscode.

Now we will start coding there. I named my directory `machine-learning-zero-to-hero` because iut's the repository for this series. So, let's get started.

Some Useful Extensions Python and ML in VS Code

- Python
- Jupyter
- Pylance
- Python Environments

Starting with Python

Make a new folder for your coding practice and open it in VS Code. Create a new file called `python_basics.ipynb` and let's start coding.

After opening the file, you will see an option to select a kernel on the top right corner of the editor. Find the `base` kernel and select it

The file should look empty but in the middle you will see `2 buttons` to 1. `Code` and 2. `Markdown`. Click on the `Code` button to start coding and you can write down your notes in the `Markdown` cells.

Like I'm doing here, Because you are reading this in a `.ipynb` file.

And now, I'll click on the `Code` button , which will create a new code cell right below this cell.

In [ ]:

```

```

Above you can see a `python` code cell, which is where you can write your python code. So, let's write our first python code.

> You can run the cell by clicking on the `Run` button on the top right corner of the cell or by pressing `Shift + Enter` on your keyboard. This will execute the code in the cell and show the output below it and also create a new cell below it.

In [1]:

```python
print('Hello Jupyter Notebook!')
```

```
Hello Jupyter Notebook!
```

Now that's done, let's start with the very basics of python.

Python Basics

In my experience, programming in any language can be broken down into a few basic concepts.

- **Input/Output**
- **Variables**
- **Arrays/Lists**
- **Conditionals**
- **Loops**
- **Functions**

A great source to learn these concepts is `Python Carsh Course 2nd Edition by Eric Matthes`. This book gave me the solid foundation I needed to start programming in python. Also the book introduced me to `machine learning`. You can find it by doing a simple google search.

> PS: Not sponsored, I just love this book and I think it is a great resource for beginners.

So, let's get started with the very basics of python.

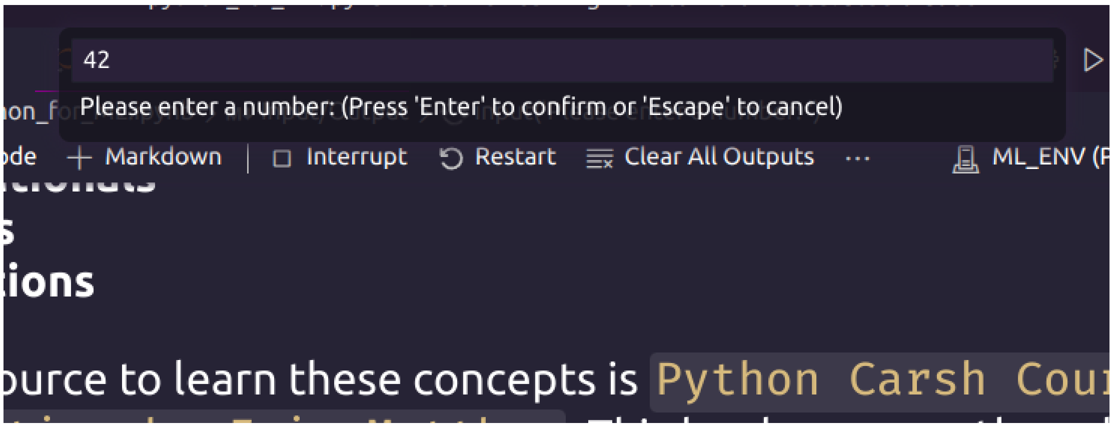
Input/Output

`Inpute/output` is the starting point of any programming language. By `input/output` I mean, how to take input from the `user` and how to `display` output to the `user`.

This happens in the `console` or `terminal` where you run your python code. In Jupyter notebooks, you can also take input from the user and display output in the same cell.

```
In [2]: # input('Please enter a number: ')
```

The `input()` function is used to take a user input. After running the cell, above you should see a pop-up asking you to enter a number in the top of the window like this:


```
42
Please enter a number: (Press 'Enter' to confirm or 'Escape' to cancel)
```

You can type anything you want and press `Enter`. It will be shown in the output below the cell like you can see above.

I entered `42` and it is shown in the output below the cell.

Python also has a built-in function called `print()` to display output to the user. You can use it to display any message.

```
In [3]: print('Hello!')
```

```
Hello!
```

After running the cell, above you should see `Hello!` in the output below the cell.

Althougth in `Jupyter notebooks` a output can be dispayed without using the `print()` function. You can just type any thing in the cell and it will be displayed in the output below the cell.

```
In [4]: 42
```

```
Out[4]: 42
```

As you can see, I just typed `42` in the cell and it is displayed in the output below the cell. This is a feature of Jupyter notebooks that makes it easy to display output without using the `print()` function at all. So, this is what we will be using to display output in our code cells. We might not use the `print()` function at all in this article series.

Now let's talk about another cruicial concept in input/output, which is `data types`.

Data Types

`Data types` are the building blocks of any programming language. They define the `type of data` that can be stored in a program. Like take the `input()` function for example.

When you take input from the user in python, it is always stored as a `string` data type. This means that the input is treated as a `sequence of characters` and not as a number. So, if you enter a number like `42`, it is stored as characters and not as a number.

So, in general, we can `add` two numbers togather but now as python treats any input as a string, if we try to add two numbers that are stored as strings, it will not work as expected.

Python will try to `concatenate` the two strings instead of adding them. For example, if you enter `42` and `58`, it will return `4258` instead of `100`.

Python has `3 main data types`:

- **String**: A sequence of characters enclosed in single or double quotes. For example, `'Hello'` or `"World"`.
- **Integer**: A whole number without a decimal point. For example, `42`
- **Float**: A number with a decimal point. For example, `3.14`

You can check the data type of a variable using the `type()` function. For example, if you want to check the data type of the input you took from the user, you can do it like this:

```
In [5]: type(input('Please enter something: '))
```

```
Out[5]: str
```

The output is `str` which means that the input is stored as a string data type.

Here is the how every type is represented in python:

```
In [6]: print('Integer:', type(42))
        print('Float:', type(42.0))
        print('String:', type('Hello'))
```

```
Integer: <class 'int'>
Float: <class 'float'>
String: <class 'str'>
```

For integer:

- You can type any whole number without a decimal point. For example, `42` or `-100`. It'll be considered as an integer data type.

For float:

- You can type any number with a decimal point. For example, `3.14`

  > If you type a number with a decimal point, it will most definitely be considered as a float data type. even if it is a whole number like `42.0`.

For string:

- You can type any sequence of characters enclosed in `single` or `double` quotes. For example, `'Hello'` or `"World"` or even `"42"` or `'3.14'`. It will be considered as a string data type.

Now you might ask, bruh but what if I want to take a number as input and use it as a number?

Well, you can convert the input to a number using the `int()` or `float()` function. But first we need to talk about `variables`.

Variables

`Variables` are used to `store data` in a program. They are like `containers` that hold a value. You can think of them as `labels` that you can use to refer to a value.

When we take a user input, we need to use those inputs in some if not all of our code. So, we need to store the input in a variable.

You can create a variable by assigning a value to it using the `=` operator. For example, if you want to store the input in a variable called `user_input`, you can do it like this:

In [7]:
```
# name = input('Please enter your name: ')
# name
```

Nice!

We took a name for the variable(name) which is refered to as `declaration` or we can say we for the code above,

We declared a variable called `name` and assigned(`=`) the value of the input to it.

I put "ITVAYA" as the input, so the variable `name` now holds the value "ITVAYA".

And when I write it in the last line of the cell, it will display the value of the variable `name` in the output below the cell.

> `=` is called the `assignment operator` in python. It is used to assign a value to a variable.

So, when ever we need a new variable, we can just declare it like this and assign any value to it.

Now, let's say we want to take a number as input and use it as a number. We can do that by converting the input to an integer or a float using the `int()` or `float()` function.

In [8]:
```python
num = int(input('Please enter a number: '))
num
```

Out[8]:
```
42
```

We just convert the input to an integer using the `int()` function and store it in a variable called `num`. Now, we can use this variable as a number in our code.

Now, if we check the type of the variable `num`, it will return `int` which means that the input is stored as an integer data type.

In [9]:
```python
type(num)
```

Out[9]:
```
int
```

If we convert the `num` variable to a float using the `float()` function, it will return `float` which means that the input is stored as a float data type.

In [10]:
```python
num = float(num)
type(num)
```

Out[10]:
```
float
```

Pretty Clear, right?

And yeah that is how we can take input from the user and use it as a number in our code.

And I think you understand the concept of variables and input/output now.

Arrays/Lists

`Arrays/Lists` are used to store a collection of data. They are like `containers` that hold a sequence of values. You can think of them as `labels` that you can use to refer to a sequence of values.

In [11]:
```python
arr = [1, 2, 3]
arr
```

```
Out[11]: [1, 2, 3]
```

Here I'm declaring a variable called `arr` and assigning it a list of values `[1, 2, 3]`. Now, we can use this variable as a list in our code.

This group of integers is called an `array` or a `list` in python.

And as we can access the list through the `arr` variable, we can also access every single element of the list using the `index` of the element. For example, if we want to access the first element of the list, we can do it like this:

```
In [12]: arr[0]
```

```
Out[12]: 1
```

The indexing starts from `0` in python and in almost every single programming language. So, now if I tell you to access the last element of the list, how would you do that?

You start counting from 0 to the last element of the list right?

So, we have a list of `[1, 2, 3]` and we want to access the last element of the list. 1 is `0` th index, 2 is `1` th index and 3 is `2` th index. So, the last element of the list is `2` th index. So, we can do it like this:

```
In [13]: arr[2]
```

```
Out[13]: 3
```

And this is we can access the last element of the list using the `index` of the element.

But this can be pretty tough to do when working with large arrays. So, what we can do in be a little clever. As the index starts from 0, the last index will always be the `length of the list - 1`. So, we can do it like this:

```
In [14]: last_index = len(arr) - 1
         arr[last_index]
```

```
Out[14]: 3
```

Here, I took a variable called `last_index` and assigned it the `length of the list - 1` using the `len()` function which returns the length of the list. Now, we can use this variable as the index of the last element of the list.

A little secret for you.

Python has a better way to access the last element of the list.

It has a built-in `right indexing`.

We index the array from the left side that starts from `0` and increments by `1`. But we can also index the array from the right side that starts from `-1` and decrements by

`-1`. So, if we want to access the last element of the list or find the second last or something like that, we can do it like this:

In [15]:
```
arr[-1], arr[-2], arr[-3]
```

Out[15]:
```
(3, 2, 1)
```

Pretty noice, right?

Now what more can we do in python?

We can do slicing.

That is clicking you asked?

Slicing

This is a very powerful feature in python. It allows you to select a range of elements from an array/iterative.

> An iterable is an object that can be iterated over, such as a list, tuple, or string.

Let's say we have a list of numbers `my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`. I want to select the first three elements of the list or mid-way elements or the last three elements.

In python we can do that pretty easily.

By using the slice operator `:` we can select a range of elements from the list.

the syntax is `my_list[start_index:end_index:step_size]`

> start_index: the index of the first element to be selected

> end_index: the index of the last element to be selected.(exclusive)

> step_size: the step size between the elements

Here is an example:

In [18]:
```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print('First three elements of the list are:')
# print(my_list[:3])
print(my_list[0:3])

print('Last three elements of the list are:')
# print(my_list[-3:])
print(my_list[7:11])

print('Middle element of the list is:')
print(my_list[4:7])
print('Copy of the list is:')
# print(my_list.copy())
print(my_list[:])
```

```
First three elements of the list are:
[1, 2, 3]
Last three elements of the list are:
[8, 9, 10]
Middle element of the list is:
[5, 6, 7]
Copy of the list is:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In this example we can see that we can get an array of items from the `my_list` list from the index of `0` to the index of `2` by `my_list[0:3]`. Here the end index is not included in the list. WHich can cause some confusion, so just remember that when you are slicing the the end_index will not be included in the list.

> You want to slice from index 4 to 7 just put [4:end_index+1] or [4:8]

> By default the start and the end index are set to 0 and the length of the list respectively. So if you don't specify the start index it will start from the index 0 and if you don't specify the end index it will end at the last index of the list.

That's why we can do this

```
In [19]: my_list[:5] # this will return from index 0 to index 4
```

```
Out[19]: [1, 2, 3, 4, 5]
```

```
In [20]: my_list[5:] # returns list from index 5 to the end
```

```
Out[20]: [6, 7, 8, 9, 10]
```

We can do the same thing with the `strings`.

As strings in python are also iterable.

So, indexing and slicing works the same way.

```
In [22]: hello = 'Hello World'

         hello[:2], hello[2:]
```

```
Out[22]: ('He', 'llo World')
```

And that's slicing in python.

Cool.

Loops

`Loops` are very important in `programming`. They allow you to repeat a block of code/operations multiple times.

Like you want to find out the sum of all the numbers in a list.

In [1]:
```python
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
summ = 0
for num in nums:
    summ = num + 1

print(summ)
```

```
11
```

For Loops

In python, the for loop works differently than in other programming languages.

The syntax is.

```python
for variable in iterable:
    # do something
```

Istead of the general `for(variable; condition; operator)` in other programming languages, python takes a iterable object and iterates over it and stores each item in the `variable`.

So, if we do some like this:

In [2]:
```python
for i in [1,2,3]:
    print(i)
```

```
1
2
3
```

We should see the elements of the list printed one by one.

In this case, the variable `i` is storing the elements of the list one by one and in each loop it is printed `one by one`.

We can do many more with for loops.

Range

Range is a function heavily connected to for loops. Because it make a iterable object of number in an array.

it has a very fimiliar syntax like the slicing(start: end: step_size) syntax.

range is a built-in function in python.

> `range(start, end, step_size)`

In this function we can pass the start, end and the step_size as arguments just like the slicing.

And also the default start is 0 and the default step_size is 1.

Here is some example of range function.

```
In [4]: list(range(5))
```

```
Out[4]: [0, 1, 2, 3, 4]
```

```
In [5]: list(range(0,5))
```

```
Out[5]: [0, 1, 2, 3, 4]
```

```
In [6]: list(range(0,5,1))
```

```
Out[6]: [0, 1, 2, 3, 4]
```

Each of the above prints the same thing.

> I'm converting it to a list to show it as an array.

So, we can use range as alternative of the traditional `for loops`.

```
In [7]: for i in range(5):
            print(i)
```

```
0
1
2
3
4
```

> we can find the sum of all the numbers in a list using the `sum()` function.

There are so, many things we can with for loops.

One of the interesting thing is `List comprehension`.

List comprehension

List comprehension is a way to create a list from a list.

```
In [1]: lt = [i for i in range(5)]
        lt
```

```
Out[1]: [0, 1, 2, 3, 4]
```

Here, we are creating a list of numbers from 0 to 4. In python if we need a quick way to create a list we can use list comprehension.

The syntax is.

`list = [expression for item in iterable]`
This will make a list of `expression` for each `item` in the `iterable`.

In the above example, we set the expression to `i` and the iterable to `range(5)`. So, it takes each number from 0 to 4 and puts it in the list.

We can do the same thing like this

In [2]:
```python
list(range(5))
```

Out[2]:
```
[0, 1, 2, 3, 4]
```

Loops for 2D arrays

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```
This is a 2D array of 3 rows and 3 columns.

As we know that a for loops iterates over each element of the list.

Let's see what happend if we iterate over the 2D array.

In [1]:
```python
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

for i in matrix:
    for j in i:
        print(j)
    print()
```

```
1
2
3

4
5
6

7
8
9
```

Now, look at this, the for loops is actually iterating over the elements of the 2D array and in the variable the element that is a list itself is bieng stored.

So if we want to print every single element of the 2D array we can do it like this.

We can run another loop inside the first loop, that will iterate over the variable.

In [3]:
```python
for i in matrix:
    for j in i:
        print(j, end=' ')
    print()
```

```
1 2 3
4 5 6
7 8 9
```

Nice right?

This is called a nested loop. We can put a loop inside another loop if there are repeated work that should be done inside the first loop.

Indentation

Indentation is a very important concept in python. That you should be very careful about.

This can break your program very easily.

Indentation is the space at the beginning of a line that indicates the level of nesting.

Some wierd words I said there.

Simply put, python doesn't have the `{}` brackets like other programming languages have for blocks of codes.

Like in other programming languages, for Loop looks like this:

```
for(i = 0; i < 10; i++){
    // do something
}
```

Here, we put out the code in the `{}` brackets to differentiate it from the rest of the code.

Bu in python there is no {} brackets.

Instead it has indentation.

In python, to specify which bode is a part of the for/while loop or conditionals or a function/class, we use indentation.

Which just a space at the beginning of the line.

```
for i in range(10):
    print(i)
    print('hello')
```

> Here you can see that inside the for loop, the print statements has a indentation of 4 spaces.

And this will tell python that these two lines are part of the for loop.

In [4]:
```
for i in range(10):
    print(i)
    print('hello')
```

```
0
hello
1
hello
2
hello
3
hello
4
hello
5
hello
6
hello
7
hello
8
hello
9
hello
```

These space is called an indentation.

Now, to explore more what we can do is remove the indentations from the second print statement.

In [5]:
```python
for i in range(10):
    print(i)

print('Hello')
```

```
0
1
2
3
4
5
6
7
8
9
Hello
```

This indicates that the second print function that prints `hello` is not part of the for loop.

That's why we can see the `hello` printed at the very last of the output.

Indentation Level

Indentation level is another important concept connected to indentation.

In the previous example we can see that by removing the indentation of a line will remove the print statement from the for loop.

But what happens if we remove only a single spave infront of the print statement?

```
In [7]: for i in range(10):
            print(i)
          print('Hello')
```

```
  File <tokenize>:3
    print('Hello')
    ^
IndentationError: unindent does not match any outer indentation level
```

And we have an error. This error tells use that the indent does does match any of the indentation levels.

This means python cannot detect which block this print statement belongs to.

So, every thing that is inside any block(loops/conditions/functions/classes) should have the same amount of space at the beginning of the line.

Here is some examples

```
In [8]: for i in range(10):
            print(i)
        print('Hello')
```

```
0
1
2
3
4
5
6
7
8
9
Hello
```

Right!!

```
In [10]: for i in range(10):
             print(i)
          print('Hello')
```

```
  File <tokenize>:3
    print('Hello')
    ^
IndentationError: unindent does not match any outer indentation level
```

WRONG!!

```
In [12]: for i in range(2):
             print(i) # part of the first loop
             for j in range(5): # part of the first loop
                 print(j) # part of the second loop
             print('Hello') # part of the first loop
```

```
0
0
1
2
3
4
Hello
1
0
1
2
3
4
Hello
```

Right!!

```
In [13]: for i in range(2):
             print(i) # part of the first loop
               for j in range(5): # indentation error
                 print(j) # part of the second loop
             print('Hello') # part of the first loop
```

```
  Cell In[13], line 3
    for j in range(5): # indentation error
    ^
IndentationError: unexpected indent
```

WRONG!!

I hope You understood why it is important to use indentation in python and why you should be careful about it.

> The best practice is to use 4 spaces/tabs for indentation. This keeps the code clean and less error prone. Practice as much as you can to learn and master this.

Now, let's talk about conditionals.

Conditionals

Every Language needs a way of logical checking. Like a number is bigger than another number or a string is longer than another string.

How do we check for it?

We use Conditionals or `if-else` statements.

Simply, we use the `if` keyword to check if a condition is `True` or `False`. If it is `True`, then we execute the code inside the `if` block. If it is `False`, then we execute the code inside the `else` block.

```
if condition:
    # do something
```

```
else:
    # do something else
```

> The the condition must be true or false.

Now, when we are talking about conditionals, we need to know about `comparison operators`. Comparison operators are used to `compare` two values and return a `True` or `False` value.

I think this example will clear things up.

In [14]:
```
num = 10

if num == 10:
    print('num is 10')
else:
    print('num is not 10')
```

```
num is 10
```

In the above example I used `==` which is called the `equality operator`. It is used to check if two values are equal.

There are many more `operators` in python. And talking about it I think I should talk about types of operators and what they do.

Types of Operators

There are many types of operators in python.

- **Arithmetic Operators**: `+`, `-`, `*`, `/`, `%`, `**`, `//`
- **Assignment Operators**: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `**=`, `//=`
- **Comparison Operators**: `==`, `!=`, `<`, `>`, `<=`, `>=`
- **Logical Operators**: `and`, `or`, `not`
- **Identity Operators**: `is`, `is not`
- **Membership Operators**: `in`, `not in`
- **Bitwise Operators**: `&`, `|`, `^`, `~`, `<<`, `>>`

A lot of them right?

Well let's go One by One.

Arithmetic Operators

These are used to do mathematical operations on numbers.

In [1]:
```
#examples of arithmetic operators
num1 = 10
num2 = 5
print('num1 + num2 =', num1 + num2) # addition
print('num1 - num2 =', num1 - num2) # subtraction
print('num1 * num2 =', num1 * num2) # multiplication
print('num1 / num2 =', num1 / num2) # division
print('num1 // num2 =', num1 // num2) # floor division
print('num1 ** num2 =', num1 ** num2) # exponent
print('num1 % num2 =', num1 % num2) # modulus
```

```
num1 + num2 = 15
num1 - num2 = 5
num1 * num2 = 50
num1 / num2 = 2.0
num1 // num2 = 2
num1 ** num2 = 100000
num1 % num2 = 0
```

For fast calculations and mathematical operations we use these operators.

Assignment Operators

Assignment operators are used to assign values to variables.

We know how to take variable right?

We use `=` after a variable name and after the `=` we add any kinda of value.

Python stores that `value` in the `variable` we declared.

> `=` is called the `assignment operator` in python. It is used to assign a value to a variable.

There are other assignment operators like `+=`, `-=`, `*=`, `/=`, `%=`, `**=`, `//=`

In [2]:
```
#examples of assignment
num1 = 10
num2 = 5
num1 += num2 # num1 = num1 + num2
print('num1 =', num1)
num1 -= num2 # num1 = num1 - num2
print('num1 =', num1)
num1 *= num2 # num1 = num1 * num2
print('num1 =', num1)
num1 /= num2 # num1 = num1 / num2
print('num1 =', num1)
num1 //= num2 # num1 = num1 // num2
print('num1 =', num1)
num1 **= num2 # num1 = num1 ** num2
print('num1 =', num1)
num1 %= num2 # num1 = num1 % num2
print('num1 =', num1)
```

```
num1 = 15
num1 = 10
num1 = 50
num1 = 10.0
num1 = 2.0
num1 = 32.0
num1 = 2.0
```

Comparison Operators

These are used to compare two values and return a `True` or `False` value.

They do not assign a value to a variable, but their sole purpose is for checking if a condition is `True` or `False`.

In [1]:
```
#example of comparison
num1 = 10
num2 = 5
print('num1 > num2 =', num1 > num2) # greater than
print('num1 < num2 =', num1 < num2) # less than
print('num1 == num2 =', num1 == num2) # equal to
print('num1 != num2 =', num1 != num2) # not equal to
print('num1 >= num2 =', num1 >= num2) # greater than or equal to
print('num1 <= num2 =', num1 <= num2) # less than or equal to
```

```
num1 > num2 = True
num1 < num2 = False
num1 == num2 = False
num1 != num2 = True
num1 >= num2 = True
num1 <= num2 = False
```

As you can see every comparison operator returns a `True` or `False` value.

And do remember what I said about the conditionals?

I'll be back there but here is some examples of other operators.

Logical Operators

Logical operators are used to `combine` multiple conditions into a single condition.

There are three logical operators in Python:

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

and

The `and` operator returns `True` if both the conditions are `True`. Otherwise, it returns `False`.

conditon 1	conditon 2	AND
True	True	True
True	False	False
False	True	False
False	False	False

or

The `or` operator returns `True` if at least one of the conditions is `True`. Otherwise, it returns `False`.

conditon 1	conditon 2	OR
True	True	True
True	False	True
False	True	True
False	False	False

not

The `not` operator returns the opposite of the condition. If the condition is `True`, it returns `False`. If the condition is `False`, it returns `True`.

conditon	NOT
True	False
False	True

In [1]:
```python
#examples of logical operators
print('AND Operator')
print(f'True and True: {True and True}')
print(f'True and False: {True and False}')
print(f'False and True: {False and True}')
print(f'False and False: {False and False}')
print()

print('OR Operator')
print(f'True or True: {True or True}')
print(f'True or False: {True or False}')
print(f'False or True: {False or True}')
print(f'False or False: {False or False}')
print()

print('NOT Operator')
print(f'not True: {not True}')
print(f'not False: {not False}')
```

```
AND Operator
True and True: True
True and False: False
False and True: False
False and False: False

OR Operator
True or True: True
True or False: True
False or True: True
False or False: False

NOT Operator
not True: False
not False: True
```

It becomes very useful when we have to deal with large dataset queries.

Identity Operator & Membership Operator

In python, there are some very special operators that are used to check if an object is equal to another object or if an object is a member of a set.

- `is` operator: This operator is used to check if `two objects` are completely equal. It is used to check if two variables are pointing to the same object in memory. That's why it is called the `identity operator`.
- `in` operator: This operator is used to check if an object is a member of a set. It is used to check if an object is present in a set.

In [3]:
```python
num1 = [1, 2, 3]
num2 = [1, 2, 3]

print(num1 is num2)
```

```
False
```

Here you can see even though `num1` and `num2` looks exactly same, but they are not stored in the same memory location. So, when we check the `num1 is num2` it will return `False` because they are stored in different memory locations.

> Here if we used the `==` operator, it will return `True` because their values are exactly same.

In [4]:
```python
num1 = [2,3,1]

print(10 in num1)
```

```
False
```

Here I'm using the `in` operator to check if `10` id in the list `[2,3,1]`. If it is in the list, it will return `True`, otherwise it will return `False`.

Bitwise Operators

I'll keep this for you guys to recap on your own. Because it's the same as the `logical` operators but they are applied on the bits of the numbers.

- `&` : Bitwise AND
- `|` : Bitwise OR
- `^` : Bitwise XOR
- `~` : Bitwise NOT

etc.

Now, that we know about the operators. It is time we see how we can use these operators in `Python`.

If Statements

`if` statements are used to execute a block of code if a certain condition is true. The syntax of an `if` statement is as follows:

In [5]:
```python
num = 10

if num > 5:
    print("num is greater than 5")

if num < 5:
    print("num is less than 5")

if num == 10:
    print("num is equal to 10")
```

```
num is greater than 5
num is equal to 10
```

As you can see, there are 3 if statement and the statement checking `num < 5` is not executed necause num is greater than 5. So, it return's `False` and the `if` statement is skipped.

if-else

`if-else` is a conditional statement that allows you to execute a `block of code` if a condition is `True` and execute another `block of code` if the condition is `False`.

Let's take the previous example and add an `else` statement to it:

In [7]:
```python
num = 10

if num < 5:
    print("less than 5")
else:
    print("greater than 5")
```

```
greater than 5
```

Here you can see the `if` statement is getting false so the code inside the `if` block is not getting executed instead as there is an `else` block which is getting executed.

if-elif-else

There might be a situation where multiple conditions need to be checked and we want to execute different code blocks based on the conditions. In such cases, we can use the `if-elif-else` statement.

In [8]:
```python
#Example of if-elif-else
num = 10

if num > 0:
    print("positive")
elif num < 0:
    print("negative")
```

```
else:
    print("zero")
```

```
positive
```

We can add numtiple elif conditions to check as many conditions as we want.

In [9]:
```
num = 32

if num < 10:
    print("num is less than 10")
elif num < 20:
    print("num is less than 20")
elif num < 30:
    print("num is less than 30")
elif num % 2 == 0:
    print("num is even")
else:
    print("num is odd")
```

```
num is even
```

> `%` operator returns the modulo of two numbers

Here's some more examples of how we can use the `conditionals`

Finding the maximum of 3 numbers

In [1]:
```
a = 3
b = 2
c = 4

if a > b and a > c:
    print("a is the largest")
elif b > a and b > c:
    print("b is the largest")
else:
    print("c is the largest")
```

```
c is the largest
```

Even or odd?

In [3]:
```
# num = int(input("Enter a number: "))
num = 35

if num % 2 == 0:
    print("Even")
else:
    print("Odd")
```

```
Odd
```

Maximum of a list

In [4]:
```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
ans = 0

for i in nums:
    if i > ans:
        ans = i

print(ans)
```

```
10
```

While Loop

There is another conditional statement in programming which doesn't really get recognised as a `conditional statement` as frequently used in programming languages. It is called `while loop` and it is a loop that executes a block of code as long as a condition is true. It's like a continuous `IF` statement.

In [5]:
```
c = 0

while c <= 10:
    print(c)
    c += 1
```

```
0
1
2
3
4
5
6
7
8
9
10
```

While loop is conditional looping, for loop is iterative looping.

While loop gives you much more control over the looping process. But it takes longer lines of code.

Now, I've talk about almost all the concepts, just a another concept to talk about.

FUNCTIONS

Functions are a way to group code together. It's like a reusable pocket of code. When we write long programs, we tend to write a lot of code in a single file and some times we write the same code over and over again.

For those, we can make a function. A function is a block of code that can be called multiple times. It can be passed arguments and it can return a value.

In [6]:
```
def func():
    print('This is a function')

func()
```

```
This is a function
```

Inside a function we can write any code we want. The function will be executed every time we call it.

But sometimes we might have to send data to it and it will give some data back. That's where the return statement and parameters/arguments come in.

In [7]:
```python
def func(param):
    print(f'this function got {param} as a parameter')

func(1)
func(34)
```

```
this function got 1 as a parameter
this function got 34 as a parameter
```

Here, I've set the parameter of the function to `param` and now python knows that this `function` needs a variable called `param` to work.

And when calling the function, we `have` to pass a `value` to the `param` variable. This is called a `argument`.

> Parameter is a simple variable set for a cirtain function to work properly. Argument is a `value/variable` passed to a function if it has any parameterd set. The value we send as a argument will the set as the value of the parameter.

But what about multiple parameters?

Multiple Parameters

If a function has multiple parameters, we can pass multiple arguments to it and we have to be little careful because we need to pass the arguments in the same order as the parameters.

In [8]:
```python
def example_func(param1, param2):
    print(f"Param1: {param1}, Param2: {param2}")

example_func(1, 2)
```

```
Param1: 1, Param2: 2
```

If we interchange the arguments the whole function can break and lead us to some very nasty bugs.

In [9]:
```python
def details(name, age):
    print(f"Name: {name}, Age: {age}")


details("John", 30)
```

```
Name: John, Age: 30
```

In [10]:
```python
details(30, 'John')
```

```
Name: 30, Age: John
```

As you can see, it changes the whole outcome of the code. And this can be a very troublesome issue if we have a function with 5 or 6 parameters.

So, for this, scenario, python has a solution called **keyword arguments**.

> Keyword arguments are a way to pass arguments to a function call that is not positional. the syntax is `func(param1=value1, param2=value2)`.

This we don't have to keep track of the order of the parameters and pass them in any order we want.

In [11]:
```python
def details(name, age, height, school):
    print(f"Name: {name}, Age: {age}, Height: {height}, School: {school}")


details(
    name="John Doe",
    height=175,
    school="MIT",
    age=20,
)
```

```
Name: John Doe, Age: 20, Height: 175, School: MIT
```

Now, what about if a function gives us something back?

This is why there is a return statement.

A return statement is used to return a value from a function.

A function can only have one return statement.

> When a function returns something the function exection is over even if the function has more statements.

Let's sa y we have a array of numbers and we want to get a array of only the even numbers from the array.

In [12]:
```python
def even(arr):
    new_arr = []
    for i in arr:
        if i % 2 == 0:
            new_arr.append(i)
    return new_arr

new_arr = even([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print(new_arr)
```

```
[2, 4, 6, 8, 10]
```

We can see that the function is returning a array of even values from the array sent as an argument. And when the function end it returns a new_array. If a function returns something it can be stored in a variable when the function is called.

That's why I did `new_arr = even([1,2,3,4,5,6,7,8,9,10])`. As the function is fully executed the returned value is stored in the variable new_arr.

and then I can just print.

And that's it for the Functions recap.

One last thing before I go.

Built-in Functions

Python has a lot of and I mean a LOT of `built-in functions`. These are functions that are already built into the language.

Here's some useful ones.

- `print()`: prints the value passed as an argument.
- `len()`: returns the length of the value passed as an argument.
- `type()`: returns the type of the value passed as an argument.
- `help()`: opens the help documentation for the function passed as an argument.
- `dir()`: returns a list of all the functions and variables in the current scope.
- `vars()`: returns a dictionary of all the variables in the current scope.
- `id()`: returns the memory address of the value passed as an argument.
- `max()`: returns the maximum value in the list passed as an argument.
- `min()`: returns the minimum value in the list passed as an argument.
- `sum()`: returns the sum of all the values in the list passed as an argument.
- `any()`: returns True if any of the values in the list passed as an argument are True.
- `all()`: returns True if all the values in the list passed as an argument are True.
- `sorted()`: returns a sorted version of the list passed as an argument.
- `abs()`: returns the absolute value of the number passed as an argument.
- `round()`: rounds the number passed as an argument to the specified number of decimal places.

Try these out yourself and experiement as much as you can.

And that's it.

Last words

For machine learning you need to be able to work with data. And there are specialized `libraries` for that and those libraries are written in `Python`. So, you need to have the basic understanding of python and programming in general. With some minor works here and there we might not use vanilla python but it is essential to understand the basics.

So, I hope this article helped you to get a recap of python and some of its basics. And we can now go into the deep deep world of `data science` and `machine learning`.

HAPPY CODING!