Comments

- Part of the code that should be skipped by the interpreter
  gives the program to write meaning full notes
- **'#'** character is used to denote comments

example:

In [10]:

```
    # this is a example
    print("hello world")
```

```
hello world
```

Note:

> it can appear at the start of a line or following white space or
> code but not within a string literal

In [11]:

```
# example:
    #text = "hello"
    text = "hello"#
    print(text)
    text = "#hello"
    print(text)
```

```
hello
#hello
```

- A hash character within a string literal is just a hash character

Variables

Has two parts:

```
1. name and
2. value
```

- To assign we use '=' character

- Name of the variable on the left side of '=' and
  Value on the right side

In [21]:

```
# eg.
    h_bar = 1.05457e-34
```

Note:

> Variable names cannot start with a digit !
> They must start with a letter or underscore!

In [20]:

```
# eg.
    #Wrong
    2boys = 42

    #Correct
    two_boys = 42
    _two_boys = 45
```

```
  File "<ipython-input-20-6e9f32474e9a>", line 6
    2boys = 42
    ^
IndentationError: unexpected indent
```

- After a var has been defined it can be manipulated as whatever we want

In [24]:

```
# eg.
    pi = 3.14159
    h = 2* pi * h_bar
    print(h)
```

```
6.6260531326e-34
```

- All variables in python are typed
  i.e the values have certain well- defined properties that dedicate how
  they are used

- Different types have different properties
  eg.

> Intgers(eg. 1,2,0,-1,..), floating point numbers (eg. 1.5,2.0,...) are
> used for mathematics

In [12]:

```
# eg.
    dims = 3
    f_dims = 3.0
    n_bar = 1.05457e-34
```

- **Strings** (str) are helpful for textual manipulation

In [28]:

```
# eg
    label = "energy (in MeV)"
```

- **Integers** and **Strings** are sometime called as **PRECISE** type since they will exactly represent the underlying idea

- **Floats** are often refered as **Imprisise** type since it doesnt always represent the correct value

In [31]:

```
# To check the type of a variable or a literal value use type() function

# type() is a built in function
# eg
    type(h_bar)
```

Out[31]:

```
float
```

In [32]:

```
    type(42)
```

Out[32]:

```
int
```

- To covert one type to another (also called as Casting!)
- To float use **float(number)**; to int use **int(number)**

In [13]:

```
# eg.
    float(42)
```

Out[13]:

```
42.0
```

In [14]:

```
    int("28")
```

Out[14]:

28

- The above example works since the string "28" has only digits
  if it had a value that made no sense as an integer then the
  conversion would fail

In [36]:

```
# eg.
    int("world")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
cell_name in async-def-wrapper()

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

- **Float** are not really real numbers as it also has **NaN** ('Not a Number' inside it)

In [1]:

```
# eg.
    type(float('NaN'))
```

Out[1]:

float

- i.e nan is a special value of float

- **Python** is **Dynamically** typed, that means:

  1. Types are set on the variable values and not on the variable names
  2. Variable types do not need to be known before the variables are used.
  3. Variable names can change types when their values are changed.

In [2]:

```
# eg.
    x = 3
    x = 5.5
    x = "hello world"
#are all valid assignments
```

- **Statically** typed languages, such as C,C++,java have:

    1. Types are set on the variable names and not on the variable values
    2. Variable types must be specified (declared or inferred) before they are used.
    3. Variable types can never change, even if the value changes.

- If a var is **not defined** i.e assigning a value, then trying to use it will give an error

In [3]:

```
# eg.
    n
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
cell_name in async-def-wrapper()

NameError: name 'n' is not defined
```

- **Note:**

> In interactive mode, the last printed expression is assgined to the variable is also used as a throwaway variable, in cases when we dont want the return value.

In [3]:

```
# eg.
    for _ in range(2):
        print("Hello")

    def foo():
        return 4,5,6,7

    a,b,_,_ = foo()
    print(a,b,_,_)
```

```
Hello
Hello
4 5 7 7
```

Boolean Values

- The values **True** and **False** makes up the entirety of the **bool type**

- It is used for:
  1.To represent truth value of python expression
  2.As flags for turing behavior on or off

- Often datas can be converted into booleans

- if the value is zero or the container is empty,
  then it is converted to False
- else if the values is non zero or non empty in any way,
  then it is converted to True

In [29]:

```
# eg.
    bool(0)
```

Out[29]:

```
False
```

In [5]:

```
    bool("hi")
```

Out[5]:

```
True
```

None

- It is a special variable in python that is used to denote that no value was given or that no behavior was defined.

- Zero is a valid number, while None is not.
- If **None** happens to make it to a point in a Program that excepts an integer or float, then the program will rightfully break.
- But with a **zero**, the program would have continued on.

- It is same as **NULL** in C/C++ and **null** in js.

Numeric Type

- int,float,long and complex

> 1.Plain integer:- long in C (8 bytes)
> 2.long integer:- unlimited precission
> 3.floating point no:- double in C (10 bytes)
> 4.complex no:- real and imaginary
>
> > It is of the form **com = a+bj**.
> > where a is real number and b is the imaginary part
> >
> > To excess real part we use **com.real**
> > To excess img part we use **com.imag**

In [6]:

```
# eg.
    com = 5 + 6j
    # excess the real part
    print("real:",com.real)
    # excess the img part
    print("img :",com.imag)
```

```
real: 5
img : 6.0
```

Operators

- used to express common ways to manipulate data and variables

- There are three classes of Operators:

> 1.Unary :- (Operates on 1 data/Variable)
> 2.Binary :- (Operates on 2 datas/Variables)
> 3.Ternary :- (Operates on 3 datas/Variables)

Unary Operators

```
Positive              +x          For numeric types, returns x.

Negative              -x          For numeric types, returns -x.

Negation           not x          Logical negation; True becomes False an
d vice versa.

Bitwise       Invert ~x          Changes all zeros to ones and vice vers
a in x’s binary representation.

Deletion           del x          Deletes the variable x.

Call                 x()          The result of x when used as a functio
n.

Assertion       assert x          Ensures that bool(x) is True.
```

Binary Operators

```
Assignment                  x = y       Set the name x to the value of y.

Attribute Access              x.y       Get the value of y which lives on t
he variable x.

Attribute Deletion     del x.y       Remove y from x.

Index                        x[y]       The value of x at the location y.

Index Deletion        del x[y]       Remove the value of x at the locati
on y.

Logical And             x and y       True if bool(x) and bool(y) are Tru
e, False otherwise.

Logical Or             x or y x       if bool(x) is True, otherwise the v
alue of y.
```

Arithmetic Binary Operators

```
Addition              x + y          The sum.

Subtraction           x - y          The difference.

Multiplication        x * y          The product.

Division              x / y          The quotient in Python 2 and true d
ivision in Python 3.

Floor Division       x // y          The quotient.

Modulo                x % y          The remainder.

Exponential          x ** y          x to the power of y.

Bitwise And           x & y          Ones where both x and y are one in
 the binary representation, zeros
                                     otherwise.

Bitwise Or            x | y          Ones where either x or y are one in
the binary representation, zeros
                                     otherwise.

Bitwise Exclusive Or  x ^ y          Ones where either x or y but not bo
th are one in the binary
                                     representation, zeros otherwise.

Left Shift           x << y          Shifts the binary representation of
x up by y bits. For integers
                                     this has the effect of multiplying
 x by 2y.

Right Shift          x >> y          Shifts the binary representation of
x down by y bits. For integers
                                     this has the effect of dividing x b
y 2y.

In-Place            x op= y          For each of the above operations, o
p may be replaced to create a
                                     version which acts on the variable
 ‘in place’. This means that
                                     the operation will be performed and
the result will immediately
                                     be assigned to x. For example, x +=
1 will add one to x.
```

Comparison Binary Operators

```
Equality                x == y       True or False.

Not Equal               x != y       True or False.

Less Than                x < y       True or False.

Less Than or Equal      x <= y       True or False.

Greater Than             x > y       True or False.

Greater Than or Equal   x >= y       True or False.

Containment             x in y       True if x is an element of y.

Non-Containment     x not in y       False if x is an element of y.

Identity Test           x is y       True if x and y point to the same un
derlying value in memory.

Not Identity Test   x is not y       False if x and y point to the same u
nderlying value in memory
```

Ternary Operator

```
Ternary Assignment       x = y = z      Set x and y to the value of z.

Attribute Assignment       x.y = z      Set x.y to be the value of z.

Index Assignment          x[y] = z      Set the location y of x to be th
e value of z.

Ternary Compare          x < y < z      True or False, equivalent to (x
 < y) and (y < z). The < here may
                                        be replaced by >, <=, or >= in a
ny permutation.

Ternary Or           x if y else z      x if bool(y) is True and z other
wise. It’s equivalent to the
                                        C/C++ syntax y?x:z.
```

- Most of the operators can be composed with one another

In [32]:

```
# eg.
    x = h = y = z= f = m =1
    x < 1 or ((h+y -f) << (m//8)) if y and z**2 else 42
```

Out[32]:

```
1
```

- **Note:**

> Certain class of operators "=" and "del"composition is not possible, because they directly modify the variables they are working with, rather than simply using their values.

- **expression**: expression is a snippet of code that does not require its own line to be executed.

In [33]:

```
# eg.
    23 * 45
```

Out[33]:

```
1035
```

- **statement**: If an operator is not fully composed and requiries its own line to work, then it is a statement

In [34]:

```
# eg.
    x = (23 * 45) + 1
```

In [ ]:

```

```