In [4]:
```
2+4
```

Out[4]: 6

In [5]:
```
%whos
```

```
Interactive namespace is empty.
```

In [6]:
```
z*x*y
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[6], line 1
----> 1 z*x*y

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

In [7]:
```
3*3
```

Out[7]: 9

In [8]:
```
2 **4
```

Out[8]: 16

In [9]:
```
2*2*2*2
```

Out[9]: 16

In [10]:
```
print(2+2)
```

```
4
```

In [11]:
```
3*3

4+4

2*2*2*2
```

Out[11]: 16

In [12]:
```
print(3+3)

print(4+4)

print (2*2*2*2)
```

```
6
8
16
```

In [13]:
```
# hashtags are COMMENTS in CODE CELLS!!!

## This is a comment

### This is still a comment!!!
```

```
### 2+2

### print (3+3)

### nothing is evaluated here!!!
```

In [14]:
```
# Comments are absolutely CRITICAL in scripts!!

# We need to use comments in order to describe what we're doing!!

### Print the result of 2+2 to the screen using the print () function

print(2+2)
```

```
4
```

This is NOT a code cell

We cannot run commands, evaluate expression, or call function

We are typing in a Markdowm text although "looks" like code... will be RENDERED in a nice format!

2+2

print (2+2)

This is not a comment

This is still not a comment

This is still not a comment

more hashtags mean smaller font

Hashtags in Markdown cells create SELECTION HEADERS!!!

More hashtags represent more sub sections

Our first notebook

This is our first Jupyter notebook. We can type Markdown text to create nice looking discussions which describe WHAT we are programming, WHY we are programming, and describe the results!!!

A simple code cell

Below is a simple code cell prints the result 2+2

In [15]:
```
print(2+2)
```

```
4
```

Bulleted lists

we can create bulleted lists using Markdown!

- Fisrt item in the list
- Second item in the list
- 3rd
- 4th
- so on and so on

back to regular working text.

Define variables

Let's define two variables, `x` and `y`, in the code cell below.

`: back ticks

In [16]:
```
x=3

y=7
```

In [17]:
```
print(x)
```

```
3
```

In [18]:
```
print(y)
```

```
7
```

In [19]:
```
print(x+y)
```

```
10
```

Jupyter notebook includes MAGIC COMMANDS that allow us to SEE and INSPECT aspects of the variables/objects in the environment!

In [20]:
```
%whos
```

```
Variable   Type    Data/Info
----------------------------
x          int     3
y          int     7
```

In [21]:
```
z=x/y
```

In [22]:
```
%whos
```

```
Variable   Type     Data/Info
-----------------------------
x          int      3
y          int      7
z          float    0.42857142857142855
```

Data types

There are many data types in programming languages like python. Data types dictate the kinds of operations we can do on the object!

The `type` function is a function that returns the DATA TYPE of the input argument.

In [23]:
```
type(x)
```

Out[23]:
```
int
```

In [24]:
```
print(type(x))
```

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

In [25]:
```
type(y)
```

Out[25]:
```
int
```

In [26]:
```
type(z)
```

Out[26]:
```
float
```

In [27]:
```
print(type(z))
```

```
<class 'float'>
```

In [28]:
```
z
### z is the decimal point
```

Out[28]:
```
0.42857142857142855
```

In [29]:
```
x+z
```

Out[29]:
```
3.4285714285714284
```

In [30]:
```
type(x+z)

### numbers integers and floats can work together
```

Out[30]:
```
float
```

In [31]:
```
x/z
```

Out[31]:
```
7.0
```

In [32]:
```
z*x
```

Out[32]:
```
1.2857142857142856
```

In [33]:
```
type(z*x)
```

Out[33]:
```
float
```

Strings

A string data type is a FUNDANMENTLLY different class than INTEGERS and FLOATS.

Strings are characters, so think of letters, words and phrases.

In [34]:
```
my_string = 'abcde'
```

In [35]:
```
%whos
```

```
Variable    Type     Data/Info
-----------------------------
my_string   str      abcde
x           int      3
y           int      7
z           float    0.42857142857142855
```

In [36]:
```
type(my_string)
```

Out[36]: str

In [37]:
```
type("abcde")
```

Out[37]: str

The `=` operator ASSIGNS values to object but the `==` operator is a CONDITIONAL or LOGICAL test about if the VALUES of the objects are the same!

In [38]:
```
'abcde'=="abcde"
```

Out[38]: True

In [39]:
```
my_string == "abcde"
```

Out[39]: True

In [40]:
```
type (my_string == "abcde")
```

Out[40]: bool

A BOOLEAN data type is a TRUE/FALSE value. Python dispaly Booleans ad `True` or `False`.

Strings are kind of "special" in that they can be SUBSET.

In [41]:
```
len(my_string)
```

Out[41]: 5

Integers do NOT have length in Python, but strings do have length in Python.

In [42]:
```
len(x)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[42], line 1
----> 1 len(x)

TypeError: object of type 'int' has no len()
```

We can SLICE or INDEX or SUBSET the character in a STRING!!!!!

The `[]` allow us to INDEX or access elements from within the STRING!

In [43]:
```
my_string[1]
```

Out[43]: 'b'

In [44]:
```
my_string
```

Out[44]: 'abcde'

VERY IMPORTANT: Python is a ZEROD BASED INDEX language!!!

In [45]:
```
my_string[0]
```

Out[45]: 'a'

Slicing

We can extract out MORE than just a single element!

We can extract or SLICE multiple elements at a time!

We slice by including the `:` operator.

The `:` operator has a START or BEGINNING index to the LEFT and an END or FINISH index to the RIGHT.

VERY IMPORTANT: the beginning index is INCLUSIVE but the ending index is EXCLUSIVE!!!!!!

In [46]:
```
my_string[0:3]
```

Out[46]: 'abc'

In [47]:
```
len(my_string[0:3])
```

Out[47]: 3

In [48]:
```
my_string[3]
```

Out[48]: 'd'

In [49]:
```
my_string[1:3]
```

Out[49]: 'bc'

A shortcut to start from the beginning to NOT include anything to the left `:`

In [50]:
```
my_string[:3]
```

Out[50]: 'abc'

In [51]:
```
my_string[:2]
```

Out[51]: 'ab'

In [52]:
```
my_string[:3]
```

Out[52]: 'abc'

In [53]:
```
my_string[:4]
```

Out[53]: 'abcd'

If we exclude the ENDING index or we do NOT include anything to the right of `:` we will return EVERYTHING from the BEGINNING to the END!

In [54]:
```
my_string[1:]
```

Out[54]: 'bcde'

In [55]:
```
my_string[2:]
```

Out[55]: 'cde'

In [56]:
```
my_string[:]
```

Out[56]: 'abcde'

If the index is NEGATIVE that starts from the END!

In [57]:
```
my_string[1]
```

Out[57]: 'b'

In [58]:
```
my_string[-1]
```

Out[58]: 'e'

We can identify the LAST element as `[-1]`

In [59]:
```
my_string[-2]
```

Out[59]: 'd'

List

This is a CONTAINER! It contains multiple elements.

Lists are initialized or CREATED with `[]` (square brackets).

In [60]:
```
an_empty_list=[]
```

In [61]:
```
%whos
```

```
Variable        Type     Data/Info
-----------------------------------
an_empty_list   list     n=0
my_string       str      abcde
x               int      3
y               int      7
z               float    0.42857142857142855
```

In [62]:
```
len(an_empty_list)
```

Out[62]: 0

In [63]:
```
type(an_empty_list)
```

Out[63]: list

Let's now make a new list that contains 4 strings.

In [64]:
```
the_hobbits =['frodo', 'sam', 'merry', 'pippin']

### `,` comma
### `:` colon
### `#` hashtag
```

In [65]:
```
len(the_hobbits)
```

Out[65]: 4

In [66]:
```
type(the_hobbits)
```

Out[66]: list

In [67]:
```
%whos
```

```
Variable        Type     Data/Info
-----------------------------------
an_empty_list   list     n=0
my_string       str      abcde
the_hobbits     list     n=4
x               int      3
y               int      7
z               float    0.42857142857142855
```

Lists like strings can be indexed, sliced, subset.

In [68]:
```
the_hobbits[0]
```

Out[68]: 'frodo'

In [69]:
```
### sliced

the_hobbits[:2]
```

Out[69]: ['frodo', 'sam']

In [70]:
```
the_hobbits
```

Out[70]: ['frodo', 'sam', 'merry', 'pippin']

In [72]:
```
the_hobbits[1]
```

Out[72]: 'sam'

In [73]:
```
the_hobbits[-1]
```

Out[73]: 'pippin'

In [74]:
```
type(the_hobbits[0])
```

Out[74]: str

In [75]:
```
type('frodo')
```

Out[75]: str

In [76]:
```
type(the_hobbits)
```

Out[76]: list

In [78]:
```
print(the_hobbits[0])

print(the_hobbits[1])

print(the_hobbits[2])

print(the_hobbits[3])
```

```
frodo
sam
merry
pippin
```

In [79]:
```
print(type(the_hobbits[0]))

print(type(the_hobbits[1]))

print(type(the_hobbits[2]))

print(type(the_hobbits[3]))
```

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

In [81]:
```
%whos
```

```
Variable        Type     Data/Info
---------------------------------
a_hobbit        str      pippin
an_empty_list   list     n=0
my_string       str      abcde
the_hobbits     list     n=4
x               int      3
y               int      7
z               float    0.42857142857142855
```

Iterate with for-loops

A for-loop aloows us to APPLY an action or EXECUTE a procedure OVER and OVER and OVER again.

We do NOT need to manually apply the action or copy/paste!!!!!

A for-loop iterates over a SEQUENCE by changing the value of an ITERATING VARIABLE along that sequence.

In [80]:
```
for a_hobbit in the_hobbits:
    print(a_hobbit)

# the cursor is not located on the far left of the cell
# the cursor is intented
```

```
frodo
sam
merry
pippin
```

In [82]:
```
%whos
```

```
Variable        Type     Data/Info
---------------------------------
a_hobbit        str      pippin
an_empty_list   list     n=0
my_string       str      abcde
the_hobbits     list     n=4
x               int      3
y               int      7
z               float    0.42857142857142855
```

In [83]:
```
the_hobbits[-1]
```

Out[83]:
```
'pippin'
```

In [85]:
```
a_hobbit
```

Out[85]:
```
'pippin'
```

A common way to create of define a for-loop is with a `range()` function.

In [86]:
```
range(4)
```

Out[86]:
```
range(0, 4)
```

In [88]:
```
print (0)

print(1)

print(2)

print(3)
```

```
0
1
2
3
```

In [89]:
```
for n in range(4):
    print(n)
```

```
0
1
2
3
```

In [90]:
```
%whos
```

```
Variable        Type     Data/Info
-----------------------------------
a_hobbit        str      pippin
an_empty_list   list     n=0
my_string       str      abcde
n               int      3
the_hobbits     list     n=4
x               int      3
y               int      7
z               float    0.42857142857142855
```

A `range()` function is often used to GENERATE a SEQUENCE of integers to subset lists!

In [91]:
```
for n in range(4):
    print (the_hobbits[n])
```

```
frodo
sam
merry
pippin
```

In [92]:
```
for a_hobbit in the_hobbits:
    print(a_hobbit)
```

```
frodo
sam
merry
pippin
```

We do not need to HARD CODE the 4 in `range()`. We can use the `len()` function to give us the LENGTH of the list!

In [93]:
```
len(the_hobbits)
```

Out[93]: 4

In [95]:
```
for n in range(len(the_hobbits)):
    print (the_hobbits[n])
```

```
frodo
sam
merry
pippin
```

Lists are heterogenous

Lists can CONTAIN MULTIPLT DATA TYPES!!!!!!

In [96]:
```
another_list=[1, 1.0, 'one', "1", '1.0', True]
```

In [97]:
```
another_list
```

Out[97]:
```
[1, 1.0, 'one', '1', '1.0', True]
```

In [98]:
```
type(another_list)
```

Out[98]:
```
list
```

In [99]:
```
%whos
```

```
Variable        Type     Data/Info
-----------------------------------
a_hobbit        str      pippin
an_empty_list   list     n=0
another_list    list     n=6
my_string       str      abcde
n               int      3
the_hobbits     list     n=4
x               int      3
y               int      7
z               float    0.42857142857142855
```

In [100…
```
len(another_list)
```

Out[100…
```
6
```

In [102…
```
for an_element in another_list:
    print(type(an_element))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'bool'>
```

In [103…
```
type(another_list[0])
```

Out[103…
```
int
```

In [104…
```
type(another_list[1])
```

Out[104…
```
float
```

In [105…
```
type(another_list[-1])
```

Out[105…
```
bool
```

In [ ]:
```

```