# Python Tutorial <https://www.kaggle.com/learn/python>

spam\_amount = 0

print(spam\_amount)

*# Ordering Spam, egg, Spam, Spam, bacon and Spam (4 more servings of Spam)*

spam\_amount = spam\_amount + 4

if spam\_amount > 0:

print("But I don't want ANY spam!")

viking\_song = "Spam " \* spam\_amount

print(viking\_song)

OUTPUT:

**0**

**But I don't want ANY spam!**

**Spam Spam Spam Spam**

 Let's review the code from top to bottom:-

spam\_amount = 0

**Variable assignment:** Here we create a variable called spam\_amount and assign it the value of 0 using =, which is called the assignment operator.

print(spam\_amount)

**Function calls:** print is a Python function that displays the value passed to it on the screen. We call functions by putting parentheses after their name, and putting the inputs (or arguments) to the function in those parentheses.

*# Ordering Spam, egg, Spam, Spam, bacon and Spam (4 more servings of Spam)*

spam\_amount = spam\_amount + 4

The first line above is a **comment**. In Python, comments begin with the # symbol. Next we see an example of reassignment. In this case, the value we're assigning to spam\_amount involves some simple arithmetic on its previous value. When it encounters this line, Python evaluates the expression on the right-hand-side of the = (0 + 4 = 4), and then assigns that value to the variable on the left-hand-side.

if spam\_amount > 0:

print("But I don't want ANY spam!")

viking\_song = "Spam Spam Spam"

print(viking\_song)

\*/ Python is prized for its readability and the simplicity/\*

Note how we indicated which code belongs to the **if**. "But I don't want ANY spam!" is only supposed to be printed if spam\_amount is positive. But the later code (like print(viking\_song)) should be executed no matter what. How do we (and Python) know that?

The colon (:) at the end of the if line indicates that a new **code block** is starting. Subsequent lines which are **indented** are part of that code block.

This code snippet is also our first sighting of a **string** in Python:

"But I don't want ANY spam!"

Strings can be marked either by double or single quotation marks. (But because this particular string contains a single-quote character, we might confuse Python by trying to surround it with single-quotes, unless we're careful.)

viking\_song = "Spam " \* spam\_amount

print(viking\_song)

The \* operator can be used to multiply two numbers (3 \* 3 evaluates to 9), but we can also multiply a string by a number, to get a version that's been repeated that many times. Python offers a number of cheeky little time-saving tricks like this where operators like \* and + have a different meaning depending on what kind of thing they're applied to. (The technical term for this is [**operator overloading**](https://en.wikipedia.org/wiki/Operator_overloading).)

## Numbers and arithmetic in Python:-

spam\_amount = 0

type(spam\_amount)

output : INT

it's an int - short for integer.

type(19.95)

output : FLOAT

A float is a number with a decimal place.

**type()** is the second built-in function. It tells us what kind of data it this refered as **Datatype.**

Python also has us covered for the rest of the basic buttons on your calculator:

| **Operator** | **Name** | **Description** |
| --- | --- | --- |
| a + b | Addition | Sum of a and b |
| a - b | Subtraction | Difference of a and b |
| a \* b | Multiplication | Product of a and b |
| a / b | True division | Quotient of a and b |
| a // b | Floor division | Quotient of a and b, removing fractional parts |
| a % b | Modulus | Integer remainder after division of a by b |
| a \*\* b | Exponentiation | a raised to the power of b |
| -a | Negation | The negative of a |

### Order of operations as **PEMDAS** - **P**arentheses, **E**xponents, **M**ultiplication/**D**ivision, **A**ddition/**S**ubtraction.

3 + 4 \* 2

Output: 5

total\_height\_meters = (hat\_height\_cm + my\_height\_cm) / 100

print("Height in meters =", total\_height\_meters)

Output: Height in meters = 4

### Builtin functions for working with numbers:

**min** and **max**return the minimum and maximum of their arguments, respectively...

print(min(1, 2, 3))

print(max(1, 2, 3))

Output: 1

Output: 3

**abs** returns the absolute value of an argument:

print(abs(32))

print(abs(-32))

Output: 32

Output: 32

**int** and **float** can also be called as functions which convert their arguments to the corresponding type:

print(float(10))

print(int(3.33))

*#They can even be called on strings!*

print(int('807') + 1)

Output: 10.0

Output: 3

Output: 808

# Functions and Getting Help

help() displays two things:

1. The number of arguments a function takes.
2. English description about the function.

help(round)

Output:- Help on built-in function round in module builtins:

round(number, ndigits=None)

Round a number to a given precision in decimal digits.

The return value is an integer if ndigits is omitted or None. Otherwise

the return value has the same type as the number. ndigits may be negative.

round(2.3)

2

## Defining functions

Builtin functions are great, but we can only get so far with them before we need to start defining our own functions. Below is a simple example.

def least\_difference(a, b, c):

diff1 = abs(a - b)

diff2 = abs(b - c)

diff3 = abs(a - c)

return min(diff1, diff2, diff3)

This creates a function called least\_difference, which takes three arguments, a, b, and c.

Functions start with a header introduced by the**def** keyword. The indented block of code following the **:** is run when the function is called.

**return** is another keyword uniquely associated with functions. When Python encounters a return statement, it exits the function immediately, and passes the value on the right hand side to the calling context.

print(

least\_difference(1, 10, 100),

least\_difference(1, 10, 10),

least\_difference(5, 6, 7), *# Python allows trailing commas in argument lists. How nice is that?*

)

Output:- 9 0 1

Python isn't smart enough to read my code and turn it into a nice English description. However, when I write a function, I can provide a description in what's called the **docstring**.

### Docstrings

def least\_difference(a, b, c):

*"""Return the smallest difference between any two numbers*

*among a, b and c.*

*>>> least\_difference(1, 5, -5)*

*4*

*"""*

diff1 = abs(a - b)

diff2 = abs(b - c)

diff3 = abs(a - c)

return min(diff1, diff2, diff3)

help(least\_difference)

Help on function least\_difference in module \_\_main\_\_:

least\_difference(a, b, c)

Return the smallest difference between any two numbers

among a, b and c.

>>> least\_difference(1, 5, -5)

4

## Functions that don't return

What would happen if we didn't include the return keyword in our function?

def least\_difference(a, b, c):

*"""Return the smallest difference between any two numbers*

*among a, b and c.*

*"""*

diff1 = abs(a - b)

diff2 = abs(b - c)

diff3 = abs(a - c)

min(diff1, diff2, diff3)

print(

least\_difference(1, 10, 100),

least\_difference(1, 10, 10),

least\_difference(5, 6, 7),

)

None None None

Python allows us to define such functions. The result of calling them is the special value**None**. (This is similar to the concept of "null" in other languages.)

mystery = print()

print(mystery)

None

## Functions Applied to Functions

Here's something that's powerful, though it can feel very abstract at first. You can supply functions as arguments to other functions. Some example may make this clearer:

def mult\_by\_five(x):

return 5 \* x

def call(fn, arg):

*"""Call fn on arg"""*

return fn(arg)

def squared\_call(fn, arg):

*"""Call fn on the result of calling fn on arg"""*

return fn(fn(arg))

print(

call(mult\_by\_five, 1),

squared\_call(mult\_by\_five, 1),

sep='**\n**', *# '\n' is the newline character - it starts a new line*

)

5

25

 max returns the largest of its arguments. But if we pass in a function using the optional key argument, it returns the argument x that maximizes key(x) (aka the 'argmax').

def mod\_5(x):

*"""Return the remainder of x after dividing by 5"""*

return x % 5

print(

'Which number is biggest?',

max(100, 51, 14),

'Which number is the biggest modulo 5?',

max(100, 51, 14, key=mod\_5),

sep='**\n**',

)

## Booleans and Conditionals

# Booleans

Python has a type of variable called bool. It has two possible values: True and False.

x = True

print(x)

print(type(x))

True

<class 'bool'>

Rather than putting True or False directly in our code, we usually get boolean values from **boolean operators**. These are operators that answer yes/no questions. We'll go through some of these operators below.

## Comparison Operations

| Operation | Description |  | Operation | Description |
| --- | --- | --- | --- | --- |
| a == b | a equal to b |  | a != b | a not equal to b |
| a < b | a less than b |  | a > b | a greater than b |
| a <= b | a less than or equal to b |  | a >= b | a greater than or equal to b |

def can\_run\_for\_president(age):

*"""Can someone of the given age run for president in the US?"""*

*# The US Constitution says you must be at least 35 years old*

return age >= 35

print("Can a 19-year-old run for president?", can\_run\_for\_president(19))

print("Can a 45-year-old run for president?", can\_run\_for\_president(45))

Can a 19-year-old run for president? False

Can a 45-year-old run for president? True

3.0 == 3

True

'3' == 3

False

def is\_odd(n):

return (n % 2) == 1

print("Is 100 odd?", is\_odd(100))

print("Is -1 odd?", is\_odd(-1))

Is 100 odd? False

Is -1 odd? True

## Combining Boolean Values

You can combine boolean values using the standard concepts of "and", "or", and "not". In fact, the words to do this are: and, or, and not.

With these, we can make our can\_run\_for\_president function more accurate.

def can\_run\_for\_president(age, is\_natural\_born\_citizen):

*"""Can someone of the given age and citizenship status run for president in the US?"""*

*# The US Constitution says you must be a natural born citizen \*and\* at least 35 years old*

return is\_natural\_born\_citizen **and** (age >= 35)

print(can\_run\_for\_president(19, True))

print(can\_run\_for\_president(55, False))

print(can\_run\_for\_president(55, True))

False

False

True

# Conditionals

Booleans are most useful when combined with conditional statements, using the keywords if, elif, and else.

Conditional statements, often referred to as if-then statements, let you control what pieces of code are run based on the value of some Boolean condition. Here's an example:

def inspect(x):

if x == 0:

print(x, "is zero")

elif x > 0:

print(x, "is positive")

elif x < 0:

print(x, "is negative")

else:

print(x, "is unlike anything I've ever seen...")

inspect(0)

inspect(-15)

0 is zero

-15 is negative

The if and else keywords are often used in other languages; its more unique keyword is elif, a contraction of "else if". In these conditional clauses, elif and else blocks are optional; additionally, you can include as many elif statements as you would like.

Note especially the use of colons (:) and whitespace to denote separate blocks of code. This is similar to what happens when we define a function - the function header ends with :, and the following line is indented with 4 spaces. All subsequent indented lines belong to the body of the function, until we encounter an unindented line, ending the function definition.

def f(x):

if x > 0:

print("Only printed when x is positive; x =", x)

print("Also only printed when x is positive; x =", x)

print("Always printed, regardless of x's value; x =", x)

f(1)

f(0)

Only printed when x is positive; x = 1

Also only printed when x is positive; x = 1

Always printed, regardless of x's value; x = 1

Always printed, regardless of x's value; x = 0

## Boolean conversion

We've seen int(), which turns things into ints, and float(), which turns things into floats, so you might not be surprised to hear that Python has a bool() function which turns things into bools.

print(bool(1)) *# all numbers are treated as true, except 0*

print(bool(0))

print(bool("asf")) *# all strings are treated as true, except the empty string ""*

print(bool(""))

*# Generally empty sequences (strings, lists, and other types we've yet to see like lists and tuples)*

*# are "falsey" and the rest are "truthy"*

True

False

True

False

# Lists

Lists in Python represent ordered sequences of values. Here is an example of how to create them:

primes = [2, 3, 5, 7]

We can put other types of things in lists:

planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']

We can even make a list of lists:

hands = [

['J', 'Q', 'K'],

['2', '2', '2'],

['6', 'A', 'K'], *# (Comma after the last element is optional)*

]

*# (I could also have written this on one line, but it can get hard to read)*

hands = [['J', 'Q', 'K'], ['2', '2', '2'], ['6', 'A', 'K']]

A list can contain a mix of different types of variables:

my\_favourite\_things = [32, 'raindrops on roses', help]

*#(Yes, Python's help function is \*definitely\* one of my favourite things)*

## Indexing

You can access individual list elements with square brackets.

Which planet is closest to the sun? Python uses zero-based indexing, so the first element has index 0

planets[0]

'Mercury'

What's the next closest planet?

planets[1]

'Venus'

Which planet is *furthest* from the sun?

Elements at the end of the list can be accessed with negative numbers, starting from -1:

planets[-1]

'Neptune'

planets[-2]

'Uranus'

## Slicing

What are the first three planets? We can answer this question using slicing:

planets[0:3]

['Mercury', 'Venus', 'Earth']

planets[0:3] is our way of asking for the elements of planets starting from index 0 and continuing up to *but not including* index 3.

The starting and ending indices are both optional. If I leave out the start index, it's assumed to be 0. So I could rewrite the expression above as:

planets[ :3]

['Mercury', 'Venus', 'Earth']

If I leave out the end index, it's assumed to be the length of the list.

planets[3:]

['Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']

i.e. the expression above means "give me all the planets from index 3 onward".

We can also use negative indices when slicing:

*#All the planets except the first and last*

planets[1:-1]

*#The last 3 planets*

planets[-3:]

['Saturn', 'Uranus', 'Neptune']

## Changing lists

Lists are "mutable", meaning they can be modified "in place".

One way to modify a list is to assign to an index or slice expression.

For example, let's say we want to rename Mars:

planets[3] = 'Malacandra'

print(planets)

['Mercury',

'Venus',

'Earth',

'Malacandra',

'Jupiter',

'Saturn',

'Uranus',

'Neptune']

Let's compensate by shortening the names of the first 3 planets.

planets[:3] = ['Mur', 'Vee', 'Ur']

print(planets)

*# That was silly. Let's give them back their old names*

planets[:4] = ['Mercury', 'Venus', 'Earth', 'Mars',]

## List functions

Python has several useful functions for working with lists.

1. len gives the length of a list:

*# How many planets are there?*

len(planets)

8

1. sorted returns a sorted version of a list:

*# The planets sorted in alphabetical order*

sorted(planets)

['Earth', 'Jupiter', 'Mars', 'Mercury', 'Neptune', 'Saturn', 'Uranus', 'Venus']

1. sum does what you might expect:

primes = [2, 3, 5, 7]

sum(primes)

17

1. We've previously used the min and max to get the minimum or maximum of several arguments. But we can also pass in a single list argument.

max(primes)

7

Interlude: objects

I've used the term 'object' a lot so far - you may have even read that *everything* in Python is an object. What does that mean?

In short, objects carry some things around with them. You access that stuff using Python's dot syntax.

For example, numbers in Python carry around an associated variable called imag representing their imaginary part. (You'll probably never need to use this unless you're doing some very weird math.)

x = 12

*# x is a real number, so its imaginary part is 0.*

print(x.imag)

*# Here's how to make a complex number, in case you've ever been curious:*

c = 12 + 3j

print(c.imag)

0

3.0

The things an object carries around can also include functions. A function attached to an object is called a **method**. (Non-function things attached to an object, such as imag, are called *attributes*).

For example, numbers have a method called bit\_length. Again, we access it using dot syntax:

x.bit\_length()

4

## List methods

list.append modifies a list by adding an item to the end:

*# Pluto is a planet darn it!*

planets.append('Pluto')

list.pop removes and returns the last element of a list:

planets.pop()

'Pluto'

### Searching lists

Where does Earth fall in the order of planets? We can get its index using the list.index method.

planets.index('Earth')

2

*# Is Earth a planet?*

"Earth" **in** planets

True

## Tuples

Tuples are almost exactly the same as lists. They differ in just two ways.

**1:** The syntax for creating them uses parentheses instead of square brackets

t = (1, 2, 3)

t = 1, 2, 3 *# equivalent to above*

**2:\*\*** They cannot be modified (they are immutable)

Tuples are often used for functions that have multiple return values.

For example, the as\_integer\_ratio() method of float objects returns a numerator and a denominator in the form of a tuple:

x = 0.125

x.as\_integer\_ratio()

(1, 8)

# Loops and List Comprehensions

For and while loops, and a much-loved Python feature: list comprehensions

# Loops

Loops are a way to repeatedly execute some code. Here's an example

planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']

for planet **in** planets:

print(planet, end=' ') *# print all on same line*

Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune

The for loop specifies

* the variable name to use (in this case, planet)
* the set of values to loop over (in this case, planets)

You use the word "in" to link them together.

The object to the right of the "in" can be any object that supports iteration. Basically, if it can be thought of as a group of things, you can probably loop over it. In addition to lists, we can iterate over the elements of a tuple:

multiplicands = (2, 2, 2, 3, 3, 5)

product = 1

for mult **in** multiplicands:

product = product \* mult

product

360

You can even loop through each character in a string:

s = 'steganograpHy is the practicE of conceaLing a file, message, image, or video within another fiLe, message, image, Or video.'

msg = ''

*# print all the uppercase letters in s, one at a time*

for char **in** s:

if char.isupper():

print(char, end='')

HELLO

### range()

range() is a function that returns a sequence of numbers. It turns out to be very useful for writing loops.

For example, if we want to repeat some action 5 times:

for i **in** range(5):

print("Doing important work. i =", i)

Doing important work. i = 0

Doing important work. i = 1

Doing important work. i = 2

Doing important work. i = 3

Doing important work. i = 4

## while loops

The other type of loop in Python is a while loop, which iterates until some condition is met:

i = 0

while i < 10:

print(i, end=' ')

i += 1 *# increase the value of i by 1*

0 1 2 3 4 5 6 7 8 9

The argument of the while loop is evaluated as a boolean statement, and the loop is executed until the statement evaluates to False.

# List comprehensions

List comprehensions are one of Python's most beloved and unique features. The easiest way to understand them is probably to just look at a few examples:

squares = [n\*\*2 for n **in** range(10)]

squares

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Here's how we would do the same thing without a list comprehension:

squares = []

for n **in** range(10):

squares.append(n\*\*2)

squares

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

We can also add an if condition:

short\_planets = [planet for planet **in** planets if len(planet) < 6]

short\_planets

['Venus', 'Earth', 'Mars']

(If you're familiar with SQL, you might think of this as being like a "WHERE" clause)

Here's an example of filtering with an if condition *and* applying some transformation to the loop variable:

*# str.upper() returns an all-caps version of a string*

loud\_short\_planets = [planet.upper() + '!' for planet **in** planets if len(planet) < 6]

loud\_short\_planets

['VENUS!', 'EARTH!', 'MARS!']

People usually write these on a single line, but you might find the structure clearer when it's split up over 3 lines:

[

planet.upper() + '!'

for planet **in** planets

if len(planet) < 6

]

['VENUS!', 'EARTH!', 'MARS!']

(Continuing the SQL analogy, you could think of these three lines as SELECT, FROM, and WHERE)

The expression on the left doesn't technically have to involve the loop variable (though it'd be pretty unusual for it not to). What do you think the expression below will evaluate to? Press the 'output' button to check.

[32 for planet **in** planets]

[32, 32, 32, 32, 32, 32, 32, 32]

List comprehensions combined with functions like min, max, and sum can lead to impressive one-line solutions for problems that would otherwise require several lines of code.

For example, compare the following two cells of code that do the same thing.

def count\_negatives(nums):

*"""Return the number of negative numbers in the given list.*

*>>> count\_negatives([5, -1, -2, 0, 3])*

*2*

*"""*

n\_negative = 0

for num **in** nums:

if num < 0:

n\_negative = n\_negative + 1

return n\_negative

Here's a solution using a list comprehension:

def count\_negatives(nums):

return len([num for num **in** nums if num < 0])

Much better, right?

Well if all we care about is minimizing the length of our code, this third solution is better still

def count\_negatives(nums):

*# Reminder: in the "booleans and conditionals" exercises, we learned about a quirk of*

*# Python where it calculates something like True + True + False + True to be equal to 3.*

return sum([num < 0 for num **in** nums])

Which of these solutions is the "best" is entirely subjective. Solving a problem with less code is always nice, but it's worth keeping in mind the following lines from [The Zen of Python](https://en.wikipedia.org/wiki/Zen_of_Python):

Readability counts.  
Explicit is better than implicit.

So, use these tools to make compact readable programs. But when you have to choose, favor code that is easy for others to understand.

# Strings

One place where the Python language really shines is in the manipulation of strings. This section will cover some of Python's built-in string methods and formatting operations.

Such string manipulation patterns come up often in the context of data science work.

## String syntax

You've already seen plenty of strings in examples during the previous lessons, but just to recap, strings in Python can be defined using either single or double quotations. They are functionally equivalent.

x = 'Pluto is a planet'

y = "Pluto is a planet"

x == y

True

Double quotes are convenient if your string contains a single quote character (e.g. representing an apostrophe).

Similarly, it's easy to create a string that contains double-quotes if you wrap it in single quotes:

print("Pluto's a planet!")

print('My dog is named "Pluto"')

Pluto's a planet!

My dog is named "Pluto"

In addition, Python's triple quote syntax for strings lets us include newlines literally (i.e. by just hitting 'Enter' on our keyboard, rather than using the special '\n' sequence). We've already seen this in the docstrings we use to document our functions, but we can use them anywhere we want to define a string.

triplequoted\_hello = """hello

world"""

print(triplequoted\_hello)

triplequoted\_hello == hello

hello

world

True

The print() function automatically adds a newline character unless we specify a value for the keyword argument end other than the default value of '\n':

print("hello")

print("world")

print("hello", end='')

print("pluto", end='')

hello

world

hellopluto

## Strings are sequences

Strings can be thought of as sequences of characters. Almost everything we've seen that we can do to a list, we can also do to a string.

*# Indexing*

planet = 'Pluto'

planet[0]

'P'

*# Slicing*

planet[-3:]

'uto'

*# How long is this string?*

len(planet)

*Yes, we can even loop over them*

[char+'! ' for char **in** planet]

['P! ', 'l! ', 'u! ', 't! ', 'o! ']

But a major way in which they differ from lists is that they are immutable. We can't modify them.

planet[0] = 'B'

*# planet.append doesn't work either*

## String methods

Like list, the type str has lots of very useful methods. I'll show just a few examples here.

*ALL CAPS*

claim = "Pluto is a planet!"

claim.upper()

'PLUTO IS A PLANET!'

*# all lowercase*

claim.lower()

'pluto is a planet!'

*# Searching for the first index of a substring*

claim.index('plan')

11

### Going between strings and lists: .split() and .join()

str.split() turns a string into a list of smaller strings, breaking on whitespace by default. This is super useful for taking you from one big string to a list of words.

words = claim.split()

words

['Pluto', 'is', 'a', 'planet!']

str.join() takes us in the other direction, sewing a list of strings up into one long string, using the string it was called on as a separator.

str=['Pluto', 'is', 'a', 'planet!']

print( " ".join(str))

Pluto is a planet!

*# Yes, we can put unicode characters right in our string literals :)*

' '.join([word.upper() for word **in** words])

'PLUTO 👏 IS 👏 A 👏 PLANET!'

### Building strings with .format()

Python lets us concatenate strings with the + operator.

planet + ', we miss you.'

'Pluto, we miss you.'

If we want to throw in any non-string objects, we have to be careful to call str() on them first.

planet + ", you'll always be the " + str(position) + "th planet to me."

"Pluto, you'll always be the 9th planet to me."

This is getting hard to read and annoying to type. str.format() to the rescue.

"**{}**, you'll always be the **{}**th planet to me.".format(planet, position)

"Pluto, you'll always be the 9th planet to me."

So much cleaner! We call .format() on a "format string", where the Python values we want to insert are represented with {} placeholders.

Notice how we didn't even have to call str() to convert position from an int. format() takes care of that for us.

If that was all that format() did, it would still be incredibly useful. But as it turns out, it can do a *lot* more. Here's just a taste:

# Dictionaries

Dictionaries are a built-in Python data structure for mapping keys to values.

numbers = {'one':1, 'two':2, 'three':3}

in this case 'one', 'two', and 'three' are the **keys**, and 1, 2 and 3 are their corresponding values.

Values are accessed via square bracket syntax similar to indexing into lists and strings.

numbers['one']

1

Or to change the value associated with an existing key

numbers['one'] = 'Pluto'

numbers

{'one': 'Pluto', 'two': 2, 'three': 3, 'eleven': 11}

Python has dictionary comprehensions with a syntax similar to the list comprehensions we saw in the previous tutorial.

planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']

planet\_to\_initial = {planet: planet[0] for planet **in** planets}

planet\_to\_initial

{'Mercury': 'M',

'Venus': 'V',

'Earth': 'E',

'Mars': 'M',

'Jupiter': 'J',

'Saturn': 'S',

'Uranus': 'U',

'Neptune': 'N'}

The in operator tells us whether something is a key in the dictionary.

'Saturn' **in** planet\_to\_initial

True

A for loop over a dictionary will loop over its keys.

for k **in** numbers:

print("**{}** = **{}**".format(k, numbers[k]))

one = Pluto

two = 2

three = 3

eleven = 11

**MORE ABOUT PYTHON :-**

1. #strings are compared lexicographically, i.e. by ASCII value of the characters

#you can remeber that capital letters come before lower case ones

**print("Scaler" > "Interviewbit") # True as 'S' comes after 'I'**

**print('s' > 'S') # True**

**print("Scaler" == "Interviewbit") # False**

1. # Identity comparisons, is keyword is used

# if the compared objects are stored in the same memory location, returns true

a = "Scaler"

b = "Scaler"

print(a is b) # **True**

print(id(a)) **#140303027477296**

print(id(b)) **#140303027477296**

1. # bitwise operators

a = 3

b = 5

c = a & b

'''

Bitwise AND (&)

In result, bit is set at those positions where it is set in both the operands

011

& 101

---

001

---

'''

print(c) #001