

Python Tutorial

Variables

▼ Variables

- Variables are containers for storing data values
- Python has no command for declaring a variable, and it is created the moment a value is assigned to it
- Variable assignment is used using an equals = clause

```
number = 10  
print(number)
```

Output:

```
10
```

- Variables do not need to be declared with any particular type, and can even change type after they have been set
- Data types can be specified using casting

```
sentence = "Hello!"  
print(sentence)  
  
x = str(3)    # x will be '3'  
y = int(3)    # y will be 3  
z = float(3)  # z will be 3.0
```

Output:

```
Hello
```

Variable Manipulation

- Variables can be assigned new values/hold arithmetic

```
number = 10  
  
number = number + 10  
print(number)  
number = number/10  
print(number)
```

Output:

```
20  
2.0
```

- Strings can be manipulated as well

```
sentence = "Hello"  
sentence = sentence + sentence  
print(sentence)
```

Output:

```
HelloHello
```

Operations

Operators

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

- Using functions calls:

```
print(min(1, 2, 3))  
print(max(1, 2, 3))  
print(abs(3))  
print(abs(-3))
```

Output:

```
1  
3  
3  
3
```

- Variables type: `int` and `float`
 - `int` stands for integer. It is the most common sort of numbers dealt with in python

```
number = 10  
type(number)
```

Output:

```
int
```

- `float` is used for numbers with decimal points for greater precision. It can be used for representing weight, proportions and measurements.

```
type(20.55)
```

Output:

```
float
```

Functions

▼ Functions

Getting Help

- The `help()` function can be used to view documentation for functions built into python

```
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.
>
```

- `help()` displays:
 - The header of the function
 - A brief description of what the function does

Defining Functions

- A function is a block of code which only runs when it is called
- A header is required, and arguments if input values are necessary
- You can pass data or known parameters/arguments into a function
- Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma
- The `return` phrase is used for returning a value from the function
- A function is defined using the `def` keyword

- To call a function, use the function name followed by paranthesis

```
def least_difference(a, b, c):
    diff1 = abs(a - b)
    diff2 = abs(b - c)
    diff3 = abs(a - c)
    return min(diff1, diff2, diff3)

print(
    least_difference(1, 10, 100),
    least_difference(1, 10, 10),
    least_difference(5, 6, 7),
)
```

Output:

```
9 0 1
```

- Notice that there are function calls within the `least_difference()` function
- Some functions do not need a `return` value
- Function arguments can also be function values
- To use functions from external libraries, the `import` word is used to import the external library:

```
import math

print("pi to 4 significant digits = {:.4}".format(math.pi))
```

Output:

```
pi to 4 significant digits = 3.142
```

- External libraries can be simplified in python using the `as` term in the import statement to import it under a shorter alias

```
import math as mt

print("pi to 4 significant digits = {:.4}".format(mt.pi))
```

Output:

```
pi to 4 significant digits = 3.142
```

Booleans/Conditionals

▼ Booleans

- Python has a type of variable called `bool`. It can either hold a value of `True` or `False`

```
x = True
print(x)
```

```
print(type(x))
```

Output:

```
True
<class 'bool'>
```

Boolean Operations

Operation	Description	Property	Operation 1	Description 1
<code>a == b</code>	<code>a</code> equal to <code>b</code>	<u>Untitled</u>	<code>a != b</code>	<code>a</code> not equal to <code>b</code>
<code>a < b</code>	<code>a</code> less than <code>b</code>	<u>Untitled</u>	<code>a > b</code>	<code>a</code> greater than <code>b</code>
<code>a <= b</code>	<code>a</code> less than or equal to <code>b</code>	<u>Untitled</u>	<code>a >= b</code>	<code>a</code> greater than or equal to <code>b</code>

- When you compare two values, the expression is evaluated and Python returns the Boolean answer
- Using the boolean operations will return boolean values

```
print(3.0 == 3)
print('3' == 3)
```

Output:

```
True
False
```

- Boolean values can also be combined using the `and` or `or` clauses

```
print(True or True and False)
```

Output:

```
True
```

- Boolean values can be negated using the `not` clause
- Python also treats different variable types as boolean values using the `bool()` function

```
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(""))
```

Output:

```
True
False
True
False
```

- Almost any value is evaluated to `True` if it has some sort of content
- Any string is `True`, except empty strings
- Any number is `True`, except `0`
- Any list, tuple, set, and dictionary are `True`, except empty ones

▼ Conditionals

- Conditional statements from mathematics:

Logical Conditions

Operation	Description	Property	Operation 1	Description 1	Status
<code>a == b</code>	<code>a</code> equal to <code>b</code>	<u>Untitled</u>	<code>a != b</code>	<code>a</code> not equal to <code>b</code>	
<code>a != b</code>	<code>a</code> is not equal to <code>b</code>	<u>Untitled</u>			
<code>a < b</code>	<code>a</code> less than <code>b</code>	<u>Untitled</u>	<code>a > b</code>	<code>a</code> greater than <code>b</code>	
<code>a <= b</code>	<code>a</code> less than or equal to <code>b</code>	<u>Untitled</u>	<code>a >= b</code>	<code>a</code> greater than or equal to <code>b</code>	
<code>a > b</code>	<code>a</code> more than <code>b</code>	<u>Untitled</u>			
<code>a >= b</code>	<code>a</code> more than or equal to <code>b</code>	<u>Untitled</u>			

- The syntax used in conditionals are `if`, `elif`, and `else`.

```
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, "not a number!")

inspect(0)
inspect(-15)
inspect(45.345)
inspect(3)
```

Output:

```
0 is zero
-15 is negative
45.345 is positive
3 is positive
```

Lists, Tuples and Dictionaries

▼ Lists

- List items are ordered, changeable, and allow duplicate values
- A list is an ordered sequence of values. It can hold integers, strings, booleans or even lists!
- When we say that lists are ordered, it means that the items have a defined order, and that order will not change

- Lists are created using square brackets

```
primes = [2, 3, 5, 7]
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
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']]
```

- Indexing: accessing individual list elements using `[]`. Using the same `planets` list above:

```
print(planets[0])
print(planets[1])
print(planets[-1])
print(planets[-2])
print(planets[0:3])
```

Output:

```
Mercury
Venus
Neptune
Uranus
['Mercury', 'Venus', 'Earth']
```

- Notice using negative integers accesses the list from the back
- You can also **slice** the list using the `[0:3]` syntax, where the printed elements are `planets[0]`, `planets[1]`, `planets[2]`
- List functions:

- `len()`
- `sorted()`
- `sum()`
- `max()`

```
print(len(planets))

# The planets sorted in alphabetical order
print(sorted(planets))

primes = [2, 3, 5, 7]
print(sum(primes))
print(max(primes))
```

Output:

```
8
['Earth', 'Jupiter', 'Mars', 'Mercury', 'Neptune', 'Saturn', 'Uranus', 'Venus']
17
7
```

- Manipulating lists:

- `append()`
- `pop()`

```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
print(planets)

planets.append('Pluto')
print(planets)

planets.pop()
print(planets)
```

Output:

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

- `append()` adds data to the end of the list
 - If you add new items to a list, the new items will be placed at the end of the list.
- `pop()` returns and removes the last item on the list
- Searching in a list

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

print("Earth" in planets)
print("Moon" in planets)
```

Output:

```
True
False
```

▼ Tuples

- Tuples are like lists, but they differ in two ways:
 1. The syntax for creating a tuple uses parentheses instead of square brackets

```
t = (1, 2, 3)
```

2. They are immutable (cannot be modified)

- Tuple items are ordered, unchangeable, and allow duplicate values

▼ Dictionaries

- Dictionaries are a built-in Python data structure for mapping keys to values
- Dictionaries are used to store data values in key:value pairs

- A dictionary is a collection which is ordered*, changeable and does not allow duplicates

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

print(numbers)
print(numbers['one'])
```

Output:

```
{'one': 1, 'two': 2, 'three': 3}
1
```

- Similar syntax can be used to add another key:value pair
- It also can be used to change the value associated with any existing key

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

print(numbers)

numbers['eleven'] = 11
print(numbers)

numbers['one'] = 'Pluto'
print(numbers)
```

Output:

```
{'one': 1, 'two': 2, 'three': 3}
{'one': 1, 'two': 2, 'three': 3, 'eleven': 11}
{'one': 'Pluto', 'two': 2, 'three': 3, 'eleven': 11}
```

- Python has dictionary comprehensions with a syntax similar to list comprehensions

```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
planet_to_initial = {planet: planet[0] for planet in planets}
print(planet_to_initial)
```

Output:

```
{'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

```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
planet_to_initial = {planet: planet[0] for planet in planets}

print('Saturn' in planet_to_initial)
print('Pluto' in planet_to_initial)
```

Output:

```
True
False
```

- A `for` loop using a `format()` function can be used to display all the key:value pairs in a dictionary

```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
planet_to_initial = {planet: planet[0] for planet in planets}

for k in planet_to_initial:
    print("{} = {}".format(k, planet_to_initial[k]))
```

Output:

```
Mercury = M
Venus = V
Earth = E
Mars = M
Jupiter = J
Saturn = S
Uranus = U
Neptune = N
```

Loops

▼ Loops

- A `for` loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string)
- This is less like the `for` keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages
- With the `for` loop we can execute a set of statements, once for each item in a list, tuple, set etc
- Print all planets in a list using the `for` loop:

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

for planet in planets:
    print(planet, end=' ') # print all on same line
```

Output:

```
Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune
```

- The `for` loop specifies a variable name (`planet` in this case) and the set of values to loop over (`planets` in this case)
- Another example:

```
multiplicands = (2, 2, 2, 3, 3, 5)
product = 1
for mult in multiplicands:
    product = product * mult
```

```
print(product)
```

Output:

```
360
```

- The `range()` function can be used in a for loop as well for specifying the number of repeats

```
for i in range(5):  
    print("Increasing the value of 1 each print. i =", i)
```

Output:

```
Increasing the value of 1 each print. i = 0  
Increasing the value of 1 each print. i = 1  
Increasing the value of 1 each print. i = 2  
Increasing the value of 1 each print. i = 3  
Increasing the value of 1 each print. i = 4
```

Strings

▼ Strings

- One place where the Python language really shines is in the manipulation of strings
- 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"  
print(x == y)
```

Output:

```
True
```

- Double quotation marks can be used within the single quotation marks to represent quotation marks within a string

```
print("Pluto's a planet!")  
print('My dog is named "Pluto"')
```

Output:

```
Pluto's a planet!  
My dog is named "Pluto"
```

- You can also use a backslash `\` in a string followed by the symbol/quotation mark within a string without causing syntax errors

```
print("Pluto's a planet!")
```

Output:

```
"Pluto's a planet!"
```

Uses of the backslash character

☰ What you type...	☰ What you get	☰ example	Aa Output
<code>\'</code>	<code>'</code>	<code>'What\'s up?'</code>	<u>What's up?</u>
<code>\"</code>	<code>"</code>	<code>"That's \"cool\""</code>	<u>That's "cool"</u>
<code>\\</code>	<code>\</code>	<code>"Look, a mountain: /\\"</code>	<u>Look, a mountain: /\</u>
<code>\n</code>	{next line}	<code>"1\n2 3"</code>	<u>1 2 {next line} 3</u>

- Examples of uses:

```
hello = "hello\nworld"  
print(hello)
```

Output:

```
hello  
world
```

- Use of triple quotes in strings:

```
triplequoted_hello = """hello  
world"""  
print(triplequoted_hello)  
print(triplequoted_hello == hello)
```

Output"

```
hello  
world  
  
True
```

- The `print()` function automatically adds a new line 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=' ')
```

Output:

```
hello
world
hello pluto
```

- Strings can be thought of a sequence of characters.
- Almost everything we've seen that we can do to a list, we can also do to a string, such as indexing:

```
planet = 'Pluto'
print(planet[0])
print(len(planet))
```

Output:

```
P
5
```

- A string also has methods for manipulation, for example:

```
claim = "Pluto is a planet!"
print(claim.upper())
print(claim.lower())
```

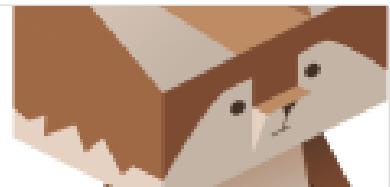
Output:

```
PLUTO IS A PLANET!
pluto is a planet!
```

- More string methods can be found here:

Python String Methods

Python has a set of built-in methods that you can use on strings. Note: All string methods returns new values. They do not change the original string. Note: All string methods returns new values. They do not change the original string. Learn more about https://www.w3schools.com/python/python_ref_string.asp



Importing and Libraries

▼ Importing

- Similar to `#include` and `"import"` in C and Java, importing is used in Python to access external libraries and modules
- One of many reasons for Python's popularity: it's vast number of libraries. There are libraries for nearly every type of coding project
- We will only be focusing on the main libraries used in data science. We will use Pandas and NumPy as examples as the rest are similar in logic.

Import the **math** library

```
import math
print(math.pi)
```

Output:

```
3.141592653589793
```

To only import the **pi** function:

```
from math import pi
print(pi)
```

Output:

```
3.141592653589793
```

Assign imports as variables for ease of use (usually used for long-named libraries)

```
import math as m
print(m.pi)
```

▼ Pandas

- This library is primarily used for importing and analyzing data at fast speeds
- Load data into something called a DataFrame

```
import pandas as pd

df = pd.DataFrame()
print(df)

lst = [10, 2, 4, 7, 12, 6]

df = pd.DataFrame(lst) #Load the data from "lst" into a pandas DataFrame
print(df)
```

Output:

```
Empty DataFrame
Columns: []
Index: []
0
0  10
1   2
2   4
3   7
4  12
5   6
```

We can see that the dataframe is at first empty, but when data is loaded into it, it transforms it into a 5×1 matrix. Here are some of the most used functionalities of Pandas:

1. Loading data from a CSV

```
import pandas as pd
df = pd.read_csv("filename.csv")
```

2. Print first 5 and last 5 lines of the data

```
import pandas as pd
df = pd.read_csv("filename.csv")
print(df.head())
print(df.tail())
```

3. Accessing/selecting data

There are two methods available: the **loc** and **iloc** methods. In a nutshell, **iloc** will simply return the data in the row that you ask it to retrieve, but only for integers. If the index/label of your data is not a number (i.e Age, Height, Weight) use **loc** to access non-integer indexed data. In this example, if your data is only numerical, row1 will equal row2 since the label of row 3 is simply '3'.

```
import pandas as pd

# making data frame from csv file
data = pd.read_csv("filename.csv")

# retrieving data in the third row
row1 = data.iloc[3]

# retrieving data whose index is 3
row2 = data.loc[3]
```

4. Convert DataFrame to NumPy array

Data will almost always have to be converted into a NumPy array before actually feeding it into a model. Use Pandas to load, format, and clean your data, then when you are ready to pass it to the model, we use NumPy

```
import pandas as pd

df = pd.read_csv("filename.csv")
df.dropna(inplace = True) #Remove any "NA" values in the dataset
arr = df.to_numpy()
```

▼ NumPy

- NumPy is a general-purpose array processing package that handles computations and manipulations of n-dimensional arrays. It is the foundation upon which most machine learning projects are built on. It is built with the ease of use of Python while utilizing the speed of C.

1. Converting an array into a NumPy array

In simple examples like these, the data is not being changed. It is simply being loaded into a different data structure: from an array into a NumPy array. When working with large amounts of data, transforming them into NumPy arrays allows for quicker computations and access to all of NumPy's functionalities.

```

import numpy as np

# Creating array object
arr = [[ 1, 2, 3],
        [ 4, 2, 5]]
arr = np.array(arr)

# Printing type of arr object
print("Array is of type: ", type(arr))

# Printing array dimensions (axes)
print("No. of dimensions: ", arr.ndim)

# Printing shape of array
print("Shape of array: ", arr.shape)

# Printing size (total number of elements) of array
print("Size of array: ", arr.size)

# Printing type of elements in array
print("Array stores elements of type: ", arr.dtype)

```

Output:

```

Array is of type: numpy.ndarray
No. of dimensions: 2
Shape of array: (2, 3)
Size of array: 6
Array stores elements of type: int64

```

2. Basic array creation and manipulation

```

import numpy as np

# Creating array from list with type float
a = np.array([[1, 2, 4], [5, 8, 7]], dtype = 'float')
print ("Array created using passed list:\n", a)

# Creating array from tuple
b = np.array((1 , 3, 2))
print ("\nArray created using passed tuple:\n", b)

# Creating a 3X4 array with all zeros
c = np.zeros((3, 4))
print ("\nAn array initialized with all zeros:\n", c)

# Create an array of complex constants
d = np.full((3, 3), 6, dtype = 'complex')
print ("\nAn array initialized with all 6s."
        "Array type is complex:\n", d)

# Create an array with random values
e = np.random.random((2, 2))
print ("\nA random array:\n", e)

# Create a sequence of integers
# from 0 to 30 with steps of 5
f = np.arange(0, 30, 5)
print ("\nA sequential array with steps of 5:\n", f)

# Create an evenly spaced sequence of 10 values in range 0 to 5
g = np.linspace(0, 5, 10)
print ("\nA sequential array with 10 values between"
        "0 and 5:\n", g)

# Reshaping 3X4 array to 2X2X3 array
arr = np.array([[1, 2, 3, 4],

```



```

        [5, 2, 4, 2],
        [1, 2, 0, 1]])

newarr = arr.reshape(2, 2, 3)

print ("\nOriginal array:\n", arr)
print ("Reshaped array:\n", newarr)

# Flatten array
arr = np.array([[1, 2, 3], [4, 5, 6]])
flarr = arr.flatten()

print ("\nOriginal array:\n", arr)
print ("Flattened array:\n", flarr)

```

Output:

```

Array created using passed list:
[[ 1.  2.  4.]
 [ 5.  8.  7.]]

Array created using passed tuple:
[1 3 2]

An array initialized with all zeros:
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]

An array initialized with all 6s. Array type is complex:
[[ 6.+0.j  6.+0.j  6.+0.j]
 [ 6.+0.j  6.+0.j  6.+0.j]
 [ 6.+0.j  6.+0.j  6.+0.j]]

A random array:
[[ 0.46829566  0.67079389]
 [ 0.09079849  0.95410464]]

A sequential array with steps of 5:
[ 0  5 10 15 20 25]

A sequential array with 10 values between 0 and 5:
[ 0.          0.55555556  1.11111111  1.66666667  2.22222222  2.77777778
 3.33333333  3.88888889  4.44444444  5.          ]

Original array:
[[1 2 3 4]
 [5 2 4 2]
 [1 2 0 1]]
Reshaped array:
[[[1 2 3]
  [4 5 2]]

 [[4 2 1]
  [2 0 1]]]

Original array:
[[1 2 3]
 [4 5 6]]
Flattened array:
[1 2 3 4 5 6]

```

3. Basic operations on an array

```

import numpy as np

a = np.array([1, 2, 5, 3])

```

```

print("Original array: ", a)

# add 1 to every element
print ("Adding 1 to every element:", a+1)

# subtract 3 from each element
print ("Subtracting 3 from each element:", a-3)

# multiply each element by 10
print ("Multiplying each element by 10:", a*10)

# square each element
print ("Squaring each element:", a**2)

# modify existing array
a *= 2
print ("Doubled each element of original array:", a)

# transpose of array
b = np.array([[1, 2, 3], [3, 4, 5], [9, 6, 0]])
print ("\nOriginal array:\n", b)
print ("Transpose of array:\n", b.T)

```

Output

```

Original array: [1 2 5 3]
Adding 1 to every element: [2 3 6 4]
Subtracting 3 from each element: [-2 -1 2 0]
Multiplying each element by 10: [10 20 50 30]
Squaring each element: [ 1  4 25  9]
Doubled each element of original array: [ 2  4 10  6]


Original array:
[[1 2 3]
 [3 4 5]
 [9 6 0]]
Transpose of array:
[[1 3 9]
 [2 4 6]
 [3 5 0]]

```

For more information on learning, looking for examples, and troubleshooting:

Python Tutorial

Well organized and easy to understand Web building tutorials with lots of examples of how to use HTML, CSS, JavaScript, SQL, PHP, Python, Bootstrap, Java and XML.

 <https://www.w3schools.com/python/default.asp>

