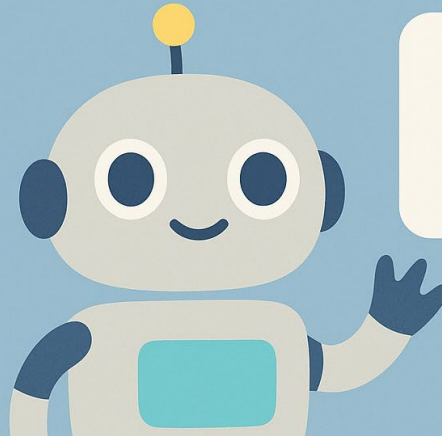


```
def greet(name):  
    print(f"Hello, {name}!")  
greet("World")
```

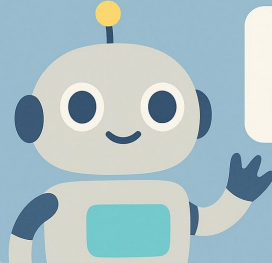
Python Basics



```
>>> python3 lesson.py  
Hello, World!
```



```
def greet(name):  
    print(f"Hello, {name}!")  
    greet("World")
```



```
>>> python3 lesson.py  
Hello, World!
```

Content

1. Data Types
2. Operators
3. Conditional Statements
4. Lists / Tuples / Dictionaries
5. Loops
6. Functions
7. Classes

1. Data Types

INTEGER

int

Positive or negative whole numbers with no decimal point

Ex:

- 1
- 23
- 1028
- -5
- 1990

FLOAT

float

Numbers with a decimal point dividing the integer and fractional parts

Ex:

- 1.0
- 1.000000
- 23.67
- 1028.34
- 1990.009

STRING

string

A sequence of characters

Ex:

- "a"
- "Hello world"
- "Maria"
- "The quick brown fox jumps over the lazy dog"

BOOLEAN

bool

Either a True or False statement

Ex:

- True
- False

1. Data Types

Any data type can be **assigned** to variables.

```
a = "Hello world"    # String
b = 10               # Integer
c = 5.5              # Float
d = True             # Boolean
```

The variables and their types can be **printed**.

```
print(a)
print(type(a))
    >> Hello world
    >> <class 'str'>
```

Variables can be **type casted** as another data type.

```
my_float = float(b)
print("Variable b type casted as: ", type(my_float))
    >> Variable b type casted as:  <class 'float'>
```

2. Operators

arithmetic operators perform basic mathematical operations on numeric data.

+, **-**, *****, **/** (addition, subtraction, multiplication, division)

****** (exponent)

% (modulus, divides left hand operand by right hand operand and returns remainder)

// (floor division, division of operands where the result is the quotient in which the digits after the decimal point are removed)

```
a = 10
b = 5

c = a + b
print(c)
>> 15

d = c / 10 + b * 5
print(d)
>> 26.5
```

```
d_square = d**2
print(d_square)
>> 702.25

print(d_square % 3)
>> 0.25

print(2 % 2)
>> 0

print(8 // 3)
>> 2

print(-8 // 3)
>> -3
```

2. Operators

arithmetic operators perform basic mathematical operations on numeric data.

+, **-**, *****, **/** (addition, subtraction, multiplication, division)

****** (exponent)

% (modulus, divides left hand operand by right hand operand and returns remainder)

// (floor division, division of operands where the result is the quotient in which the digits after the decimal point are removed)

```
a = 10
b = 5

c = a + b
print(c)
>> 15

d = c / 10 + b * 5
print(d)
>> 26.5
```

```
d_square = d**2
print(d_square)
>> 702.25

print(d_square % 3)
>> 0.25

print(2 % 2)
>> 0

print(8 // 3)
>> 2

print(-8 // 3)
>> -3
```

2. Operators

assignment operators assign a value to a variable. They might also include an arithmetic operations before assigning.

=, += Add AND, -= Subtract AND, *= Multiply AND, /= Divide AND,
%= Modulus AND, **= Exponent AND, //= Floor Division

```
a = 10
print("Assignment: ", a)
>> Assignment: 10
```

```
a += 1
print("Addition: ", a)
>> Addition: 11
```

```
a *= 3
print("Multiplication: ", a)
>> Multiplication: 30
```

```
a **= 2
print("Exponentiation: ", a)
>> Exponentiation: 100
```


2. Operators

comparison operators check if the statement is true and return **True** or **False**.

==, != or **<>, >, <, >=, <=**

Example: Is it true that "a" == "b"? Is it true that "c" is greater than "b"?

```
a = 10
b = 10
c = 11

print("a is equal to b: ", a==b)
    >> a is equal to b: True
print("a is equal to c: ", a==c)
    >> a is equal to c: False
print("a is NOT equal to b: ", a!=b)
    >> a is NOT equal to b: False

print("c is greater than b: ", c > b)
    >> c is greater than b: True
print("c is less than b: ", c < b)
    >> c is less than b: False
print("a is greater than or equal to b: ", a >= b)
    >> a is greater than or equal to b:
True
print("a is less than or equal to c: ", a <= c)
    >> a is less than or equal to c: True
```

2. Operators

logical operators combine multiple conditions together and return **True** or **False**.
and, or, not

```
print(True or False)
>> True
print(True and False)
>> False
print(10 == 10 or 3 <= 2)
>> True
print(not True)
>> False
```

A	B	A AND B	A OR B	NOT A
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

2. Operators

membership operators, identity operators check if the statement is true and return **True** or **False**.

in, not in (membership)

is, is not (identity)

Example: Is it true that "a" is "b"? Is it true that "a" is in "list_1"?

```
a = 1
b = 20

print(a is b)
>> False
print(a is not b)
>> True
```

```
a = 1
b = 20
list_1 = [1, 2, 3, 4, 5]

print(a in list_1)
>> True
print(b in list_1)
>> False
print(a not in list_1)
>> False
```

3. Conditional statements

conditional statements execute different actions based on the conditions that are set.

if a statement is True
 >> do something

elif another statement is True
 >> do something

else
 >> do something

if both statement A is True **and** statement B is True
 >> do something

if either statement A is True **or** statement B is True
 >> do something

if statement **is not** True
 >> do something

4. Lists / Tuples / Dictionaries

LIST [...]

list

Containers with multiple variables. Variables can be of any data type.

Ex:

- `number_list = [1, 3, 13, 75]`
- `string_list = ["hello", "goodbye", "no more ideas", "etc."]`
- `mixed_list = ["strings", 5, 6, 6.7, True, "etc."]`

TUPLE (...)

tuple

Similar to lists except they are immutable – they cannot be changed in size or dimension.

Ex:

- `my_tuple_1 = (1, 2, 3)`
- `my_tuple_2 = ("hello", "goodbye")`

DICTIONARY {...}

dict

Stores items in key-value pairs. Items can be appended and removed.

Ex:

- `my_dict = {"Name" : "Begum", "Age" : 88, "Profession" : "Unknown"}`

4. Lists / Tuples / Dictionaries

list item access

```
my_list = list(range(10))

print(my_list)
>> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(len(my_list))
>> 10

print(my_list[1])
>> 1

print(my_list[-5])
>> 5

print(my_list[2:5])
>> [2, 3, 4]

print(my_list[:3])
>> [0, 1, 2]

print(my_list[6:])
>> [6, 7, 8, 9]
```

Generate a list of integers starting at 0.

Read the length of a list.

Access by referring to the index number (starts at index 0)

Access beginning from the end (-1 is the last item)

Range of indexes with a start and an end (excludes end)

Range starts at the first item until the defined end.

Range starts at the defined item until the end of the list.

4. Lists / Tuples / Dictionaries

list methods

```
my_list.append("apple")
print(my_list)
>> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
'apple']

my_list.pop(2)
print(my_list)
>> [0, 1, 3, 4, 5, 6, 7, 8, 9, 'apple']

my_list.remove("apple")
print(my_list)
>> [0, 1, 3, 4, 5, 6, 7, 8, 9]

my_list.insert(1, "orange")
print(my_list)
>> [0, 'orange', 1, 3, 4, 5, 6, 7, 8, 9]
```

Appends an element at the end of the list.

Removes an element at the specified location.

Removes the first element with the specified value.

Adds an element at the specified location.

4. Lists / Tuples / Dictionaries

dict item access and methods

```
my_dict = {"Name" : "Begüm", "Age" : 88, "Profession" : "Unknown"}  
print(my_dict["Age"])  
>> 88
```

Access a value in the dictionary by its key.

```
my_dict.update({"Name" : "Olala"})  
print(my_dict)  
>> {'Name': 'Olala', 'Age': 88,  
'Profession': 'Unknown'}  
print(my_dict.keys())  
>> dict_keys(['Name', 'Age',  
'Profession'])  
print(my_dict.values())  
>> dict_values(['Olala', 88,  
'Unknown'])  
print(my_dict.items())  
>> dict_items([('Name', 'Olala'),  
( 'Age', 88), ('Profession', 'Unknown')])
```

Update a value in the dictionary by its key.

Return a list containing the dictionary's keys.

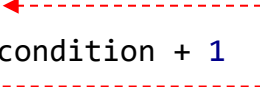
Return a list of all the values in the dictionary.

Return a list containing a tuple for each key value pair.

5. Loops

a **while-loop** is a control flow statement for specifying iteration, which allows code to be executed repeatedly.

```
break_condition = 0
```

```
while break_condition < 10: 
    break_condition = break_condition + 1
    print(break_condition)
else:
    print("out of while loop")
```

```
>> 1
```

```
>> 2
```

```
>> 3
```

```
...
```

```
>> 9
```

```
>> 10
```


```
>> out of while loop
```

5. Loops

a **for-loop** is a control flow statement for specifying iteration, which allows code to be executed repeatedly.


```
numbers = range(3)
print(numbers)
```

```
for n in numbers:
    print(n / 5.0 + 100)
```



```
>> range(0, 3)
>> 100.0
>> 100.2
>> 100.4
```

```
my_list = ["dog", "cat"]
for p, x in enumerate(my_list):
    print(p, x)
```



```
>> 0 dog
>> 1 cat
```

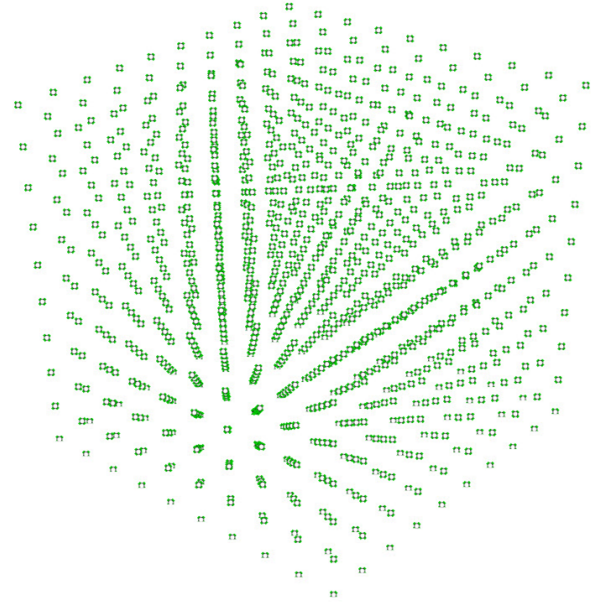
5. Loops

a **nested loop** is a loop statement inside another loop statement.

```
points = []
```

```
for i in range(0,10):  
    for j in range(0,10):  
        for k in range(0,10):  
            point = Point(i,j,k)  
            points.append(point)
```

```
a = points
```



6. Functions

a **function** is a block of code which only runs when it is called. Call that function as many times as you want with new input parameters inside your code later!

```
def my_basic_func(a, b):  
    c = a + b  
    return c
```

```
x = my_basic_func(0, 0)  
y = my_basic_func(3, 20)  
z = my_basic_func(54, 3)
```

```
print(x, y, z)  
    >> 0 23 57
```

7. Classes

a **class** a code template for creating objects. Creating a new class creates a new *type* of object. Using this template, as many instances of that type can be made. Each class instance can have attributes attached to it for maintaining its state.

```
class Vehicle():  
    def __init__(self, colour, nb_wheels, name):  
        self.colour = colour  
        self.nb_wheels = nb_wheels  
        self.name = name
```

```
vehicle_1 = Vehicle("blue", 2, "bike")  
vehicle_2 = Vehicle("red", 4, "car")
```

```
print("This is a " + vehicle_1.colour + " " + vehicle_1.name + " with ", vehicle_1.nb_wheels, " " +  
"wheels")
```

```
>> This is a blue bike with 2 wheels.
```

```
print("This is a " + vehicle_2.colour + " " + vehicle_2.name + " with " + str(vehicle_2.nb_wheels) +  
" " + "wheels")
```

```
>> This is a red car with 4 wheels.
```

7. Classes

class instances can also have methods (defined by its class) for modifying its state.

```
class Rectangle():
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def get_area(self):
        return self.length * self.width

my_rectangle = Rectangle(length=3, width=5)

area = my_rectangle.get_area()

print(area)
>> 15
```

For further information, you can look into the [Python Cheat Sheet](#).

Enjoy coding!