

# Introduction

Python is a high-level, general-purpose programming language created by Guido Van Russom and released in 1991. Its design philosophy emphasises code readability with the use of significant indentation. Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including structured, object-oriented and functional programming.

Features of Python:

- Easy to learn:  
Python is high level language that is easy to learn,
- Easy to code:  
With it's like english language it's easier to write code
- Dynamically typed:  
Unlike in other static typed programming languages such as C, C++, where you have to specify the data type that the variable is going to store.
- 

Application in:

Python Interactive shell

Installation:

Windows os

To install Python on your windows machine, go to [python.org](https://python.org) official website

Linux os

Macos

First Python program

## Python Basics

Variables:

A variable in Python is a container for storing a value. When declared it acts as a reference to that value that has been assigned to it. Or in another way we can say that when

Characteristics of a variable:

1. Act as a container for storing values.
2. It's value changes

**<variable\_name> = <value>**



```
1  # A vriable Syntax
2  <variable_name> = <value>
```

### **Examples:**



```
1  # A variable with string value ( in other words anything within double ot single quote)
2  country = "United Kingdom"
3
4
5  # A variable with integer value ( -3,-2,-1..0..1, 2, 3)
6  num1 = -10
7  num2 = 15
8
9
10 # A variable with float value
11 pi = 3.14
12
13
14 # A variable with complex number
15 complex_num = 2 + 4j
```

### **Example:**

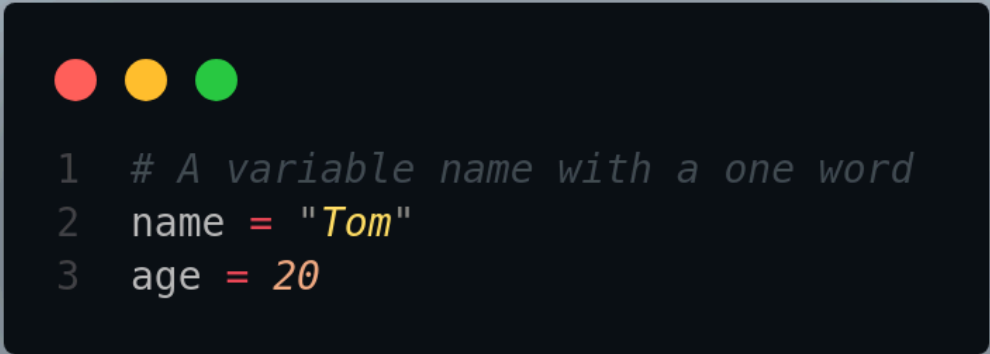
We write some variables following some patterns know as cases, such as Snake case, Camel Case:

In “snake case” the words are separated by underscore ( \_ ).

The reason it's called "snake case" is because the underscore ( \_ ) looks like the belly of the snake.

In "camelCase" words are joined together without space, with each new word starting with a capital letter.

*Example of a variable name with one word:*



```
1  # A variable name with a one word
2  name = "Tom"
3  age = 20
```

*Example of a variable with two or more words ( which is also in camelCase)*



```
1 # A name with two or more words which is also in Snake Case
2 first_name = "Tom"
3 tom_age = 20
```

### Rules for naming variable:

A variable name must:

1. Start with a letter or the underscore
2. Can not start with a number
3. Can only contain alpha-numeric characters and underscores (A-z, 0-9, and \_)
4. Variable names are case sensitive ( age, Age and AGE are three different variables)

### Keywords:

Keywords are reserved words that can not be used to name variables, functions or other identifiers because they have been reserved and they have a special meaning to the language.

as	and	or	def	del	if	elif
try	except	final	match	for	while	else
from	import	not	in	class	assert	break
continue	False	True	global	nonlocal	lambda	pass
None	return	yield	with	raise	case	

Exercises:

1. Create a variable to store the following information of the users

## Operators

### Arithmetic operators:

There are seven arithmetic operators in Python

`+, -, *, /, **, //, %`

Operator	Description	Example
<b>+</b>	<b>Addition:</b> Adds two or more operands together.	<code>a = 3 + 4</code>  <code># a = 7</code> <code>print(a)</code>
<b>-</b>	<b>Subtraction:</b> Subtracts the left operand from the right operand	<code>x = 9</code> <code>y = 3</code>  <code>print(x - y)</code>  <code># output 3</code>

<b>*</b>	<b>Multiplication:</b> Multiply the left operand by the right operand	x = 3 y = 3  print(x * y)  # output 9
<b>/</b>	<b>Division:</b> Divide the left operand by the right operand	x = 10 y = 2  print(x / y)  # output is 5.0
<b>**</b>	<b>Exponent:</b> Takes the left operand(value) as the base, also assign the right operand(value) as a power to be raised to the first operand.	x = 5 y = 2  print(x ** y)  # similar to 5 raised to power 2
<b>%</b>	<b>Modulus:</b> Divides the left operand(value) by the right operand(value) and returns the remainder.	x = 10 y = 3  print(x % y)  # output is 1
<b>//</b>	<b>Floor Division:</b> Divides the left operand(value) by the right operand(value) and returns the result without the remainder.	x = 10 y = 3  print(x // y)  # output is 3

### Assignment operators:

Generally in programming, assignment operators are those operators that are used to assign a value to a variable.

=, +=, -=, \*=, /=, //=, %=

Operator	Description	Example
=	<b>Assignment operator:</b> Assign the right value to the left variable or assign the value to a variable	x = "Python" y = x print(y)  # output 'Python'
+=		
-=		
*=		
/=		
//=		

### Comparison operators:

Comparison operators help us in writing logical instructions to the computer, and also in controlling the flow of our instructions.

==, <, >, <=, >=

Operator	Description	Example
==	<b>Equal:</b> Compares for the equality of two operands	x = 20 y = 10 z = 20  x == y # False x == z # True



<b>=!</b>	<b>Not equal:</b> Compares for inequality of the two operands. Returns <b>True</b> if the two operands are unequal and <b>False</b> if not.	x = 20 y = 10 z = 20  x != y # True x != z # False
<b>&gt;</b>	<b>Greater than:</b> Compares if the left operand(value) is greater than the right operand(value). Returns <b>True</b> if greater than and <b>False</b> if not	x = 20 y = 10 z = 20  x > y # True x > z # False
<b>&lt;</b>	<b>Less than:</b> Compares if the left operand(value) is less than the right operand(value). Returns <b>True</b> if less than and <b>False</b> if not	x = 20 y = 10 z = 30  x < y # False x < z # True
<b>&gt;=</b>	<b>Greater or equal:</b> Check if the operand(value) on the left side is greater than or equals to the operand(value) on the right side.	x = 20 y = 10 z = 20 w =  x >= y # True x >= z # True
<b>&lt;=</b>	<b>Less or equal:</b> Check if the operand(value) on the left side is less than or equals to the operand(value) on the right side.	x = 20 y = 10 z = 20  x <= y # False x <= z # True

### Membership operators:

Operator	Description	Example
<b>in</b>	Checks if the value is in the sequence or list of values. Returns <b>True</b> if the	name = "Python"

	value is in the sequence else <b>False</b> .	n = "n" print(n in name)
<b>not in</b>	Checks if the value is not in the sequence or list of values. Returns <b>True</b> if the value is not in the sequence else <b>False</b> .	

### Identity operators:

*We use Identity operators to check if the two objects aka two variables have the same value.*

Operator	Description	Example
<b>is</b>	Checks if the two objects have the same value. Returns <b>True</b> if they have the same value, else <b>False</b> .	x = 10 y = 12 z = 10
<b>is not</b>	Checks if the two objects don't have the same value. Returns <b>True</b> if they don't have the same value, else <b>False</b> .	x = 10 y = 12 z = 10

## Logical operators

*We use logical operators to check whether a certain condition is met before executing the programming.*

Operator	Description	Example
and		
or		
not		

## Bitwise operators

There are six bitwise operators in Python.

1. <<  
a << b

This is the same as multiplying a by 2\*\*b

2. >>  
a >> b

3. &
4. |
5. ~
6. ^

Data Types:

Python has several built-in data types such as simple data types like integers, float, complex

immutable data types such as integers, floats, complex, and strings can not be changed in place

In contrast, mutable data types like list, dictionaries can be modified in place, meaning their values can be changed without creating new objects.

## **Indexes:**

Python String:

## Python List:

Accessing elements in a List

Adding elements in a list

Modifying elements in a list

Methods

## Python Tuple:

Tuple is an immutable data type that is used to store a collection of data. It can be of the same or different data type.

### **Creating a tuple:**

*A tuple data type can be created using parentheses ().*

```
# creating a tuple with different data type  
student_info = ("Ben",
```

*# Incorrect way of creating a tuple with one item*  
*number = (2)*

*# Correct way of creating a tuple with one item*  
*number = (2,)*

*When creating a tuple with one item, we put a comma after that item, so*

## **Accessing items in a tuple:**

*Items can be accessed through indexing, slicing and iteration, but we'll focus on the first and second method.*

### ***Indexing:***

*items = (3, "Python", [3, 5], 'Ben')*

Item	3	"Python"	[3, 5]	'Ben'
Positive Index	0	1	2	3
Negative Index	-4	-3	-2	-1

### ***Slicing:***

*Slicing allows you to print*

## **Modifying a tuple:**

*Tuple is one of the immutable of the data type, meaning when you try to modify a tuple after you declare or create it, it's going to give you the following error:*

*TypeError: tuple doesn't support item assignment.*

*Which means once you create a tuple you can't modify the items that you have given to it.*

## **Tuple methods:**

*Methods are just functions that belong to a specific data type, so the following methods(functions) belong to a tuple data type.*

Method	Description	Example
count()	Counts how many times an item exists in a tuple.	colours = ("red", 'green', 'red', 'blue')  print(colors.count('red'))
index()	Checks at which position or index an item exists in a tuple.	colours = ("red", 'green', 'red', 'blue')  print(colors.index('red'))

## **Python Dictionary:**

Dictionary methods:

### **1. Clear()**

Removes all items(Key/value) from the current dictionary

### **2. copy()**

Creates a shallow copy of the current dictionary object,

```
student = {"name": "Ben", "age": 21}
```

```
new_student = {}
```

```
new_student = student.copy() # Creates a shallow copy of student dictionary
```

### **3. fromkeys()**

Create a dictionary with keys from iterable and values set to value.

### **4. get()**

get() takes a key as an argument and returns a value corresponding to it.

### **5. items()**

### **6. keys()**

keys returns all the keys with a dictionary.

```
student = {"name": "Ben", "age": 20, "school": "ULK"}
```

```
print(student.keys()) # will output: (['name', 'age', 'school'])
```



### **7. pop()**

Removes a value corresponding to the key it takes.

### **8. popitem()**

Removes the last key/value pairs in a dictionary.

### **9. setdefault()**

Adds a key to a dictionary and sets it's value to None.

### **10. update()**

### **11. values()**

This method is used to get all the values in a dictionary.

## **Python Sets:**

Is a collection of data types that is not ordered, mutable, iterable and does not have duplicate elements.

## **No duplicate elements**

In python the elements of a set should not be identical, meaning the elements in a single set should be unique.

## **Unordered:**

Since a set is unordered, the elements can not be referred to by index or key, because the elements do not have a defined order.

## **Mutable:**

In python a set is mutable, because we can add and remove elements in a set after creation.

## **Accessing Items in a set:**

Since set items do not have index or key we can only access its elements by traversing or looping through them.

```
fruits = {"apple", "banana", "cherry", "orange"}
```

```
for fruit in fruits:  
    print(fruit)
```

## **Adding items into a set:**

To add elements into a set use add() method:

For example:

```
my_set = {4, 6, 8}  
print(f"The original set: {my_set}")
```

```
my_set.add(10)
print(f'The updated set: {my_set}')
```

### Adding sets:

Also to add items from other set you can use the update() method:  
Ben

For example:

```
set1 = {3, 4, 5}
set2 = {6, 7, 8}
```

```
set1.update(set2)
print(set1)
```

### **Removing an item from a set:**

To remove elements from a set

```
remove(element)
```

Removes the specified element from a set.

```
pop()
```

```
clear()
```

clear() method removes all the elements of a set.

```
set1 = {3, 6, 9, 12}
print(set1) # Will returns a set containing the above elements
```

`set1.clear()` # Will remove all the elements inside a set, returns an empty set

## **Set operators:**

Subtraction operator(-):

Returns a set containing the elements in the current set object that do not exist in the other set, else it returns an empty set.

Union (|)

Combines the elements in both sets and returns a new set, containing those elements (no duplication).

```
set_x = {1, 2, 3, 4}
```

```
set_y = {4, 5, 6, 7}
```

```
set_x | set_y # Will combine the elements in both set into new set
```

```
# We can store it in a variable called new set
```

```
new_set = set_x | set_y
```

Intersection (& or `intersection()`)

It returns the elements found in both sets as a new set. Else if there is not intersection then it will return an empty set.

```
x = {3, 5, 2, 6, 7}
```

```
y = {4, 1, 3, 2, 8}
```

```
x & y # Will return the elements that are common in both sets as a new set.
```

```
# We can store it in a new set
```

```
new_set = x & y
```

```
print(new_set) # Will output: {2, 3}
```

Subset ( `<=` or `issubset()` )

Test if every element in the set is in another.

In other words, it checks if every element in the current set is in the other set.

Superset ( `>=` or `issuperset()` )

Test whether every element in the other is in the set.

In other words, it checks if every element in the other set exists or can be found in this set.

```
set_x = {5, 3, 1}
```

```
set_y = {1, 2, 3, 4, 5, 6, 7, 8}
```

```
set_y >= set_x # True: cause every element in set_x is a member in set_y but  
not every element in set_y is a member in set_x
```

Difference ( `'-'` or `difference()` )

Asymmetric difference ( `'^'` or `asymmetric_difference()` )

Returns a new set containing elements that exist in the current set and not the other also the elements that exist in the other set and don't exist in this set.

Returns the elements that exist in this set only and not the other, the same with other sets as a new set.

**Set methods:**

`difference()`

`set1 = {1, 2, 3}`

`set2 = {3, 4, 5}`

`result = set1.difference(set2)`

`print(result)`

This method calculates the difference between set1 and set2. It returns a new set containing elements that are present in set1 but not in set2.

In this case, the difference between set1 and set2 is {1, 2}. Because 1 and 2 are in set1 but not in set2.

`print(result)`: This prints the result of the difference operation

### **1. add(element)**

Adds an element to a set. When adding numbers, it makes sure that the number is being added with respect to their order, meaning if the number is less than a specific element, it will be added in front of that element else after it.

### **2. clear()**

### **3. copy()**

### **4. difference(other\_set)**

The difference method returns the difference between two sets, This method returns a new set t

```
set1 = {1, 2, 3, 4, 5}
```

```
set2 = {4, 5, 6, 7, 8}
```

```
set1.difference(set2)
```

It will return the elements in set 1 that are not in set2

## **5. union(other\_set)**

The union method returns a new set containing the elements of both sets. Meaning it combines the elements from both sets, also taking into consideration that there is no duplication of items.

```
set1 = {1, 2, 3, 4, 5}
```

```
set2 = {4, 5, 6, 7, 8}
```

```
set3 = set1.union(set2)
```

Note: If you try to combine the elements of set3 with the elements of set2 to again, it'll try to add the elements in set2 but then it will check and find out that the elements are already in set3.

```
set3.union(set1)
```

```
print(set3) # Nothing will be added because the elements in set1 already exist in set3
```

```
set3.union(set2)
```

```
print(set3) # Nothing will be added also because the elements in set2 already exist in set3
```

## **6. difference\_update(other\_set)**

Removes all the elements of another set from this set.

Simply, removes all the elements that are in this set that are also members or exist in the other set.

```
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}
```

### **7. discard(element)**

Removes an element from a set if it is a member. Unlike remove() method, it doesn't throw an error if the element is not a member of a set.

Example:

```
my_set = {1, 3, 5, 6, 7, 9}
my_set.discard(6) # Will remove 6 from my_set
print(my_set) # Will output: {1, 3, 5, 7, 9}
```

### **8. intersection(other\_set)**

Returns the intersection of two sets as a new set. In other words it returns the elements found in both sets as a new set.

```
set_x = {3, 5, 7, 8, 9}
set_y = {4, 12, 7, 5}
new_set = set_x.intersection(set_y) # Return a new set if one or more
elements exist in both sets, else return empty set
print(new_set) # Will output: {5, 7}
```

### **9. intersection\_update(other\_set)**

Update a set with intersection of itself and another. In other words it will update the previous elements of the current set with its intersection with



another, meaning remove the elements that were in this set and update this set with the intersection of itself and another set.

Example:

```
set_x = {3, 5, 7, 8, 9}
```

```
set_y = {4, 12, 7, 5}
```

```
# Since the intersection of set_x and set_y is {5, 7}
```

```
# Before intersection_update; set_x = {3, 5, 7, 8, 9}
```

```
set_x.intersection_update(set_y) # Will update set_x by its intersection with  
set_y
```

```
# After intersection_update: set_x = {5, 7}
```

## 10. isdisjoint(other\_set)

Returns True if two sets have a null intersection. This set method returns True if there is no intersection between this set and the other set, else returns False.

```
set_x = {1, 2, 3, 4, 5}
```

```
set_y = {2, 4, 6, 8, 10}
```

```
set_z = {10, 11, 12, 13}
```

```
set_x.isdisjoint(set_x) # Returns False since there is intersection between the  
two sets
```

```
set_x.isdisjoint(set_z) # Returns True since there is intersection between the  
two sets.
```

## 11. issubset(other\_set)

Test if every element in the set is in another.

In other words, it checks if every element in the current set is in the other set.

```
set_x = {1, 2, 3, 4, 5}
```

```
set_y = {1, 2, 3, 4, 10, 20}
```

```
set_z = {1, 2, 3, 4, 5, 10, 11, 12, 13}
```

`set_x.issubset(set_y)` # Returns False, it's true that 1, 2, 3, and 4 are in `set_y` but 5 has to be in `set_y` too for `set_x` to be a subset of `set_y`

`set_x.issubset(set_z)` # Return True, since all elements in `set_x` are also elements in `set_z`

## 12. `issuperset(other_set)`

Test whether every element in the other is in the set.

In other words, it checks if every element in the other set exists or can be found in this set.

```
set_x = {1, 2, 3, 4, 5}
```

```
set_y = {1, 2, 3, 4, 10, 20}
```

```
set_z = {1, 2, 3, 4, 5, 10, 11, 12, 13}
```

`set_x.issuperset(set_z)` # Returns False because not all elements in `set_z` can be found in `set_x`

`set_y.issuperset(set_z)` # Return False also because not all elements in `set_z` can be found in `set_y`

`set_z.issuperset(set_x)` # Return True since every element in `set_x` exists in `set_z`.

## 13. `pop()`

Removes an element and returns it.

```
my_set = {5, 3, 6, 4}
```

```
my_set.pop()
```

```
# output: 3  
print(my_set) # Will output: {5, 6, 4}
```

#### **14. remove(element)**

Removes an element provided from a set.

```
set_a = {1, 2, 3, 4, 5}  
  
set_a.remove(2) # removes 2 from the set  
print(set_a) # Will output: {1, 3, 4, 6}  
set_a.remove(4) # removes 4 from the set  
print(set_a) # Will output: {1, 3, 5}
```

#### **15. symmetric\_difference(other\_set)**

Return the symmetric difference of the two sets as a new set.  
(i.e. all elements that are exactly one of the sets)

Simply it returns elements that only belong to one of the sets.

```
set_x = {1, 2, 3, 4, 5, 6}  
set_y = {20, 40, 50}  
set_z = {4, 2, 9, 10, 11, 1, 13}  
  
set_x.symmetric_difference(y) # Will output a new set: {2, 3, 5, 6, 9, 11, 13}
```

#### **16. symmetric\_difference\_update(other\_set)**

Update the set with the symmetric difference of itself and another.

It's the same as `difference_update`, but instead of assigning the elements that only exist in the current set and not the other, it assigns the elements that only exist in one set and not the other to the current set.

```
set_x = {1, 2, 3, 4, 5, 6}
set_y = {20, 40, 50}
set_z = {4, 2, 9, 10, 11, 1, 13}
```

```
set_v = set_x.symmetric_difference_update(set_z)
```

## 17. `update()`

## Control flow:

## Conditional Statements:

<https://www.sciencedirect.com/topics/engineering/logical-expression#:~:text=A%20logical%20expression%20is%20a%20statement%20that%20can%20either%20be,a%20and%20b%20are%20given.>

## If statement:

If statement is used to check the expression whether it's logically true or false and then execute an instruction based on truthfulness of condition satisfied.

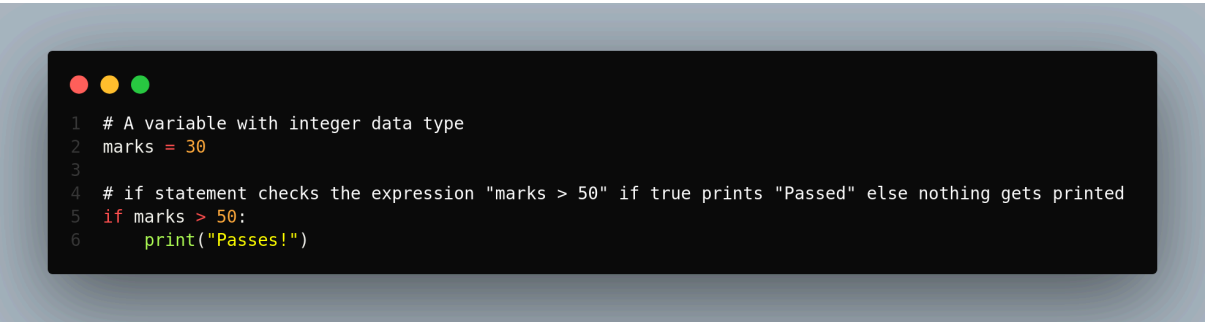
Syntax:

if expression:

    # block to be executed

if-statement checks if the expression is true, if it's true everything inside the block gets executed, if it's false then everything inside the block gets skipped.

Examples:



```
1 # A variable with integer data type
2 marks = 30
3
4 # if statement checks the expression "marks > 50" if true prints "Passed" else nothing gets printed
5 if marks > 50:
6     print("Passes!")
```

## elif statement:

elif statement in python is the same as else if statements in other programming languages such as C, C++, Java, JavaScript and C#. It is used to test multiple conditions, which follows the if statement.

Example:



```
1 x = 10
2 y = 10
3
4 if y > x:
5     print("Y is greater than X")
6 elif y == x:
7     print("Y is equal to X")
```

else statement:

match statement:

match statement in Python is similar to switch in other programming languages such as JavaScript, C, C++, C#, and Java. What it does is that it allows you to test multiple case scenarios where the condition can be true, just like the way you multiple elif is written to test multiple expressions.

Example:

1. Using elif to check for multiple case scenarios where the condition might be true.

## 2. Using match statement to

## Loop statements:

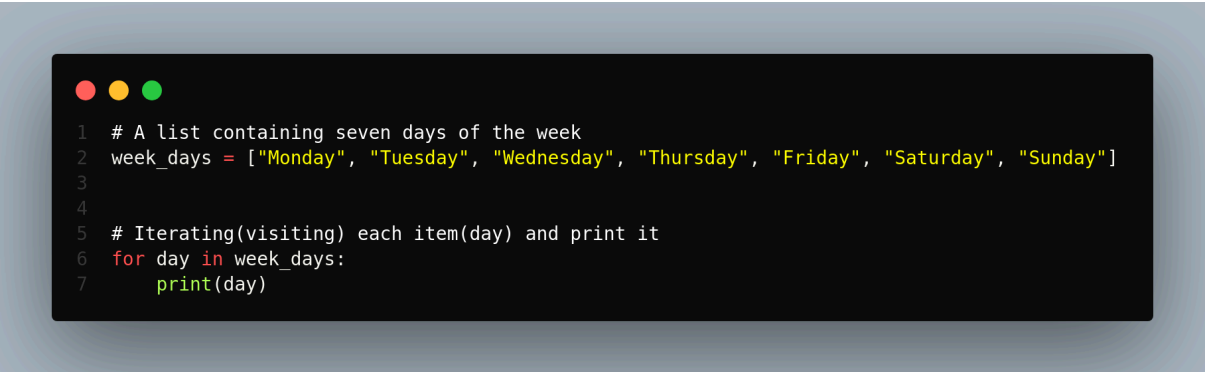
These are the statements that execute Functions: a sequence of statements many times until a condition becomes false.

### for loop:

A for loop is used to iterate over the elements of a sequence, such as string, list, tuple and range.

Syntax:

```
for element in sequence:  
    print(element)
```



```
1 # A list containing seven days of the week  
2 week_days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]  
3  
4  
5 # Iterating(visiting) each item(day) and print it  
6 for day in week_days:  
7     print(day)
```



```
1 # range function creates a collection of numbers starting from 1 - 9
2
3 for number in range(10):
4     print(number)
```

Nested for loop:

A nested for loop is simply a for nested(put inside) in another loop:

for element in elements:

    # code to be executed

    for item in items:

        # code to be executed

Exercise:

1. Create a list that has countries names as items and print each name, along with the following message:  
For example: Country: Britain
2. Create a tuple that contains a list of colours, use a for loop to iterate over the colours.



while loop:

## break and continue

The break statement terminates the execution of the program when the outer condition is met. While the continue statement is used to skip the

### **break:**

In this example we have a for loop that's supposed to loop over the numbers starting from 0 ranging to 10, but the if condition inside the for loop checks that when num is equal to 4, which is the number of iteration, the break statement should stop the iteration.

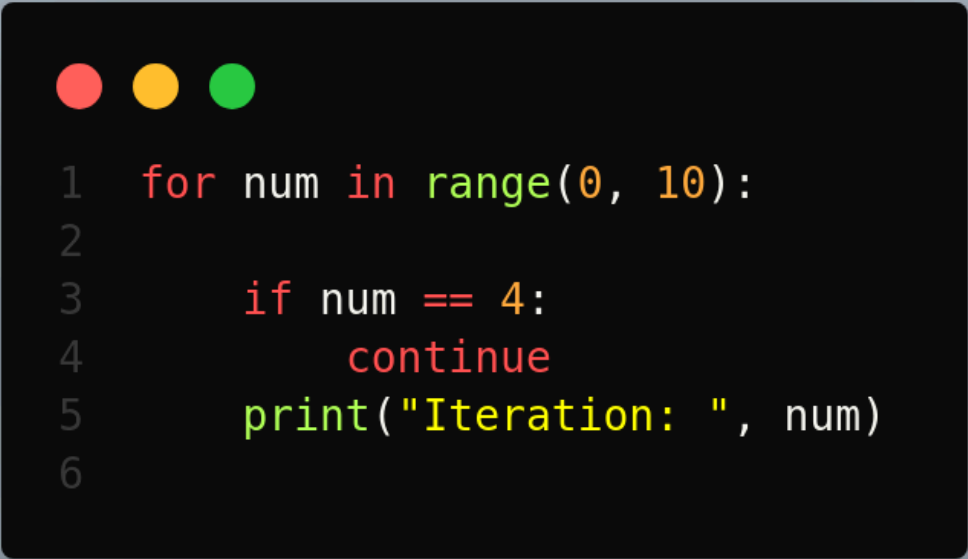


```
1  for num in range(0, 10):  
2  
3      if num == 4:  
4          break  
5      print("Iteration: ", num)
```

# continue

Continue statement is used to skip the instructions or the code inside the loop

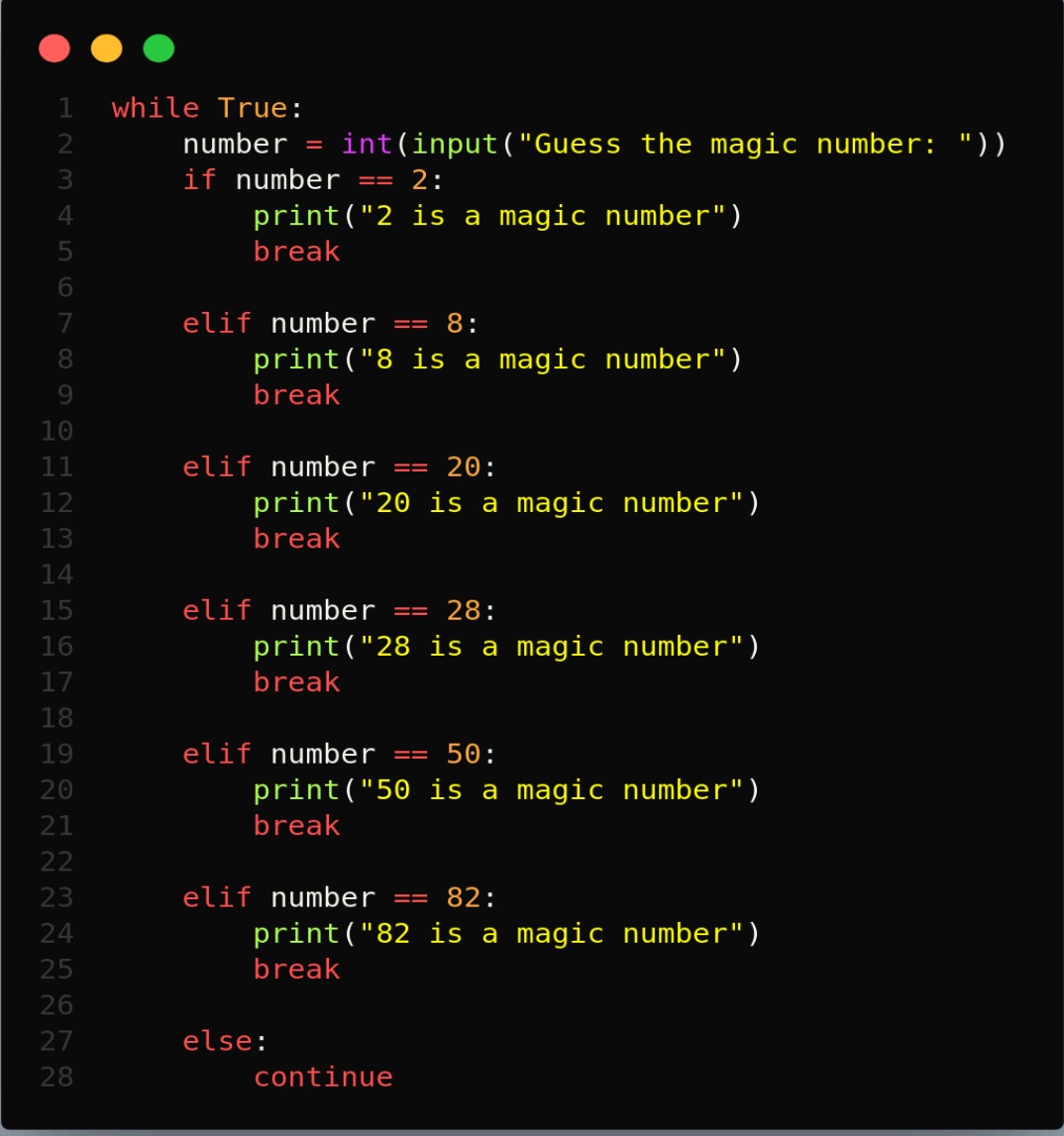
We are using the same below above, the only thing we've change is the break statement into continue statement, what continue statement does is that when the following conditions becomes true, so the for loop while run for first time where num will be equals to 0, and then num will be checked whether it's equal to 4 or not, since it's not we loop will run for the second time, where num will be equal to 1 which is not equal to 4, so when the for loop runs for the fifth time the if statement again will check if the num is equal to 4 which is true because zero is the first iteration, then the continue statement inside it will be executed because num == 4, and the continue statement will skip or jump the print("Iteration: ", num) which is supposed to display "Iteration: 4".

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) at the top left. The code is written in a syntax-highlighted style. It shows a for loop that iterates from 0 to 9. Inside the loop, there is an if statement that checks if the current number is 4. If it is, the 'continue' statement is executed, which skips the rest of the loop body and jumps to the next iteration. Otherwise, it prints the current iteration number and the value of 'num'.

```
1  for num in range(0, 10):  
2  
3      if num == 4:  
4          continue  
5      print("Iteration: ", num)  
6
```

**Combination of break and continue**

As you can see in the example below. We have a while loop whose expression is True and by that we have constructed an infinite loop which will run indefinitely until interrupted by external events.



```
1  while True:
2      number = int(input("Guess the magic number: "))
3      if number == 2:
4          print("2 is a magic number")
5          break
6
7      elif number == 8:
8          print("8 is a magic number")
9          break
10
11     elif number == 20:
12         print("20 is a magic number")
13         break
14
15     elif number == 28:
16         print("28 is a magic number")
17         break
18
19     elif number == 50:
20         print("50 is a magic number")
21         break
22
23     elif number == 82:
24         print("82 is a magic number")
25         break
26
27     else:
28         continue
```

The above program runs in the following way, the True statement starts the



Functions:





## Python Intermediate



# Python Advanced