

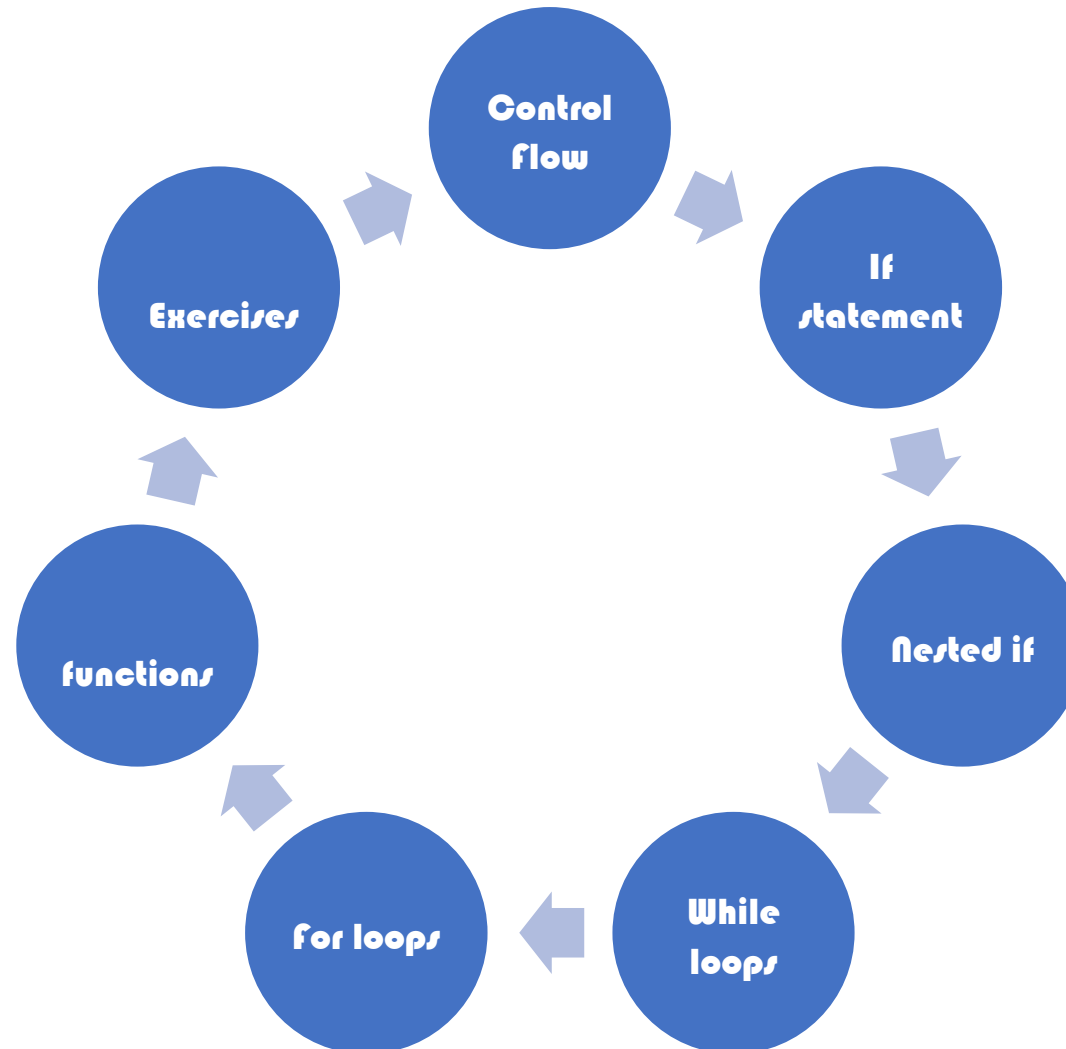


INTRODUCTION TO PYTHON PROGRAMMING

- Lecture two -

PRESENTED BY YOUSSEF KHALIL

Session Agenda



IF STATEMENT



Python Conditions and If statements

Python supports the usual logical conditions from mathematics:

- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`

These conditions can be used in several ways, most commonly in "if statements" and loops.

An "if statement" is written by using the `if` keyword.

IF STATEMENT



Example

If statement:

```
a = 33
b = 200
if b > a:
    print("b is greater than a")
```

[Try it Yourself »](#)

IF STATEMENT



Indentation

Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

Example

If statement, without indentation (will raise an error):

```
a = 33
b = 200
if b > a:
print("b is greater than a") # you will get an error
```

[Try it Yourself »](#)

IF STATEMENT



Elif

The `elif` keyword is python's way of saying "if the previous conditions were not true, then try this condition".

Example

```
a = 33
b = 33
if b > a:
    print("b is greater than a")
elif a == b:
    print("a and b are equal")
```

[Try it Yourself »](#)

IF STATEMENT



Else

The `else` keyword catches anything which isn't caught by the preceding conditions.

Example

```
a = 200
b = 33
if b > a:
    print("b is greater than a")
elif a == b:
    print("a and b are equal")
else:
    print("a is greater than b")
```

[Try it Yourself »](#)

IF STATEMENT



Short Hand If

If you have only one statement to execute, you can put it on the same line as the if statement.

Example

One line if statement:

```
if a > b: print("a is greater than b")
```

[Try it Yourself »](#)

IF STATEMENT



Short Hand If ... Else

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

Example

One line if else statement:

```
a = 2  
b = 330  
print("A") if a > b else print("B")
```

[Try it Yourself »](#)

IF STATEMENT



And

The `and` keyword is a logical operator, and is used to combine conditional statements:

Example

Test if `a` is greater than `b`, AND if `c` is greater than `a`:

```
a = 200
b = 33
c = 500
if a > b and c > a:
    print("Both conditions are True")
```

[Try it Yourself »](#)

IF STATEMENT



Or

The `or` keyword is a logical operator, and is used to combine conditional statements:

Example

Test if `a` is greater than `b`, OR if `a` is greater than `c` :

```
a = 200
b = 33
c = 500
if a > b or a > c:
    print("At least one of the conditions is True")
```

[Try it Yourself »](#)

IF STATEMENT



Nested If

You can have `if` statements inside `if` statements, this is called *nested if* statements.

Example

```
x = 41

if x > 10:
    print("Above ten,")
    if x > 20:
        print("and also above 20!")
    else:
        print("but not above 20.")
```

[Try it Yourself »](#)

IF STATEMENT



The pass Statement

`if` statements cannot be empty, but if you for some reason have an `if` statement with no content, put in the `pass` statement to avoid getting an error.

Example

```
a = 33
b = 200

if b > a:
    pass
```

[Try it Yourself »](#)

IF STATEMENT EXERCISE



CHECK THE NUMBER IS ODD OR EVEN

CHECK THE NUMBER IS POSITIVE OR NEGATIVE

WHILE LOOPS



Example

Print i as long as i is less than 6:

```
i = 1
while i < 6:
    print(i)
    i += 1
```

[Try it Yourself »](#)

WHILE LOOPS



The break Statement

With the `break` statement we can stop the loop even if the while condition is true:

Example

Exit the loop when i is 3:

```
i = 1
while i < 6:
    print(i)
    if i == 3:
        break
    i += 1
```

[Try it Yourself »](#)

WHILE LOOPS



The continue Statement

With the `continue` statement we can stop the current iteration, and continue with the next:

Example

Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
    i += 1
    if i == 3:
        continue
    print(i)
```

[Try it Yourself »](#)

The else Statement

With the `else` statement we can run a block of code once when the condition no longer is true:

Example

Print a message once the condition is false:

```
i = 1
while i < 6:
    print(i)
    i += 1
else:
    print("i is no longer less than 6")
```

[Try it Yourself »](#)

FOR 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.

Example

Print each fruit in a fruit list:

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    print(x)
```

[Try it Yourself »](#)

Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

Example

Loop through the letters in the word "banana":

```
for x in "banana":  
    print(x)
```

[Try it Yourself »](#)

The break Statement

With the `break` statement we can stop the loop before it has looped through all the items:

Example

Exit the loop when `x` is "banana":

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
    if x == "banana":
        break
```

[Try it Yourself »](#)

The continue Statement

With the `continue` statement we can stop the current iteration of the loop, and continue with the next:

Example

Do not print banana:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        continue
    print(x)
```

[Try it Yourself »](#)

The range() Function

To loop through a set of code a specified number of times, we can use the `range()` function,

The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example

Using the `range()` function:

```
for x in range(6):  
    print(x)
```

[Try it Yourself »](#)

FOR LOOPS



Example

Using the start parameter:

```
for x in range(2, 6):  
    print(x)
```

[Try it Yourself »](#)

The `range()` function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: `range(2, 30, 3)`:

Example

Increment the sequence with 3 (default is 1):

```
for x in range(2, 30, 3):  
    print(x)
```

[Try it Yourself »](#)

Else in For Loop

The `else` keyword in a `for` loop specifies a block of code to be executed when the loop is finished:

Example

Print all numbers from 0 to 5, and print a message when the loop has ended:

```
for x in range(6):  
    print(x)  
else:  
    print("Finally finished!")
```

[Try it Yourself »](#)

FOR LOOPS



Nested Loops

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":

Example

Print each adjective for every fruit:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
    for y in fruits:
        print(x, y)
```

[Try it Yourself »](#)

The pass Statement

`for` loops cannot be empty, but if you for some reason have a `for` loop with no content, put in the `pass` statement to avoid getting an error.

Example

```
for x in [0, 1, 2]:  
    pass
```

[Try it Yourself »](#)

STRINGS ARE ARRAYS



Slice To the End

By leaving out the *end* index, the range will go to the end:

Example

Get the characters from position 2, and all the way to the end:

```
b = "Hello, World!"  
print(b[2:])
```

EXERCISE TIME



USE THE LOOPS TO GET THE FACTORS OF A NUMBER

Creating a Function

In Python a function is defined using the `def` keyword:

Example

```
def my_function():  
    print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

Example

```
def my_function():  
    print("Hello from a function")  
  
my_function()
```

[Try it Yourself »](#)

Arguments

Information can be passed into functions as arguments.

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 following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

```
def my_function(fname):  
    print(fname + " Refsnes")  
  
my_function("Emil")  
my_function("Tobias")  
my_function("Linus")
```

[Try it Yourself »](#)

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Example

This function expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, lname):  
    print(fname + " " + lname)  
  
my_function("Emil", "Refsnes")
```

[Try it Yourself »](#)

FUNCTIONS



Example

This function expects 2 arguments, but gets only 1:

```
def my_function(fname, lname):  
    print(fname + " " + lname)  
  
my_function("Emil")
```

[Try it Yourself »](#)

Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a `*` before the parameter name in the function definition.

This way the function will receive a *tuple* of arguments, and can access the items accordingly:

Example

If the number of arguments is unknown, add a `*` before the parameter name:

```
def my_function(*kids):  
    print("The youngest child is " + kids[2])  
  
my_function("Emil", "Tobias", "Linus")
```

[Try it Yourself »](#)

Keyword Arguments

You can also send arguments with the *key = value* syntax.

This way the order of the arguments does not matter.

Example

```
def my_function(child3, child2, child1):  
    print("The youngest child is " + child3)  
  
my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

[Try it Yourself »](#)

Arbitrary Keyword Arguments, **kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: `**` before the parameter name in the function definition.

This way the function will receive a *dictionary* of arguments, and can access the items accordingly:

Example

If the number of keyword arguments is unknown, add a double `**` before the parameter name:

```
def my_function(**kid):  
    print("His last name is " + kid["lname"])  
  
my_function(fname = "Tobias", lname = "Refsnes")
```

[Try it Yourself »](#)

Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without argument, it uses the default value:

Example

```
def my_function(country = "Norway"):  
    print("I am from " + country)  
  
my_function("Sweden")  
my_function("India")  
my_function()  
my_function("Brazil")
```

[Try it Yourself »](#)

Passing a List as an Argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:

Example

```
def my_function(food):  
    for x in food:  
        print(x)  
  
fruits = ["apple", "banana", "cherry"]  
  
my_function(fruits)
```

[Try it Yourself »](#)

Return Values

To let a function return a value, use the `return` statement:

Example

```
def my_function(x):  
    return 5 * x  
  
print(my_function(3))  
print(my_function(5))  
print(my_function(9))
```

[Try it Yourself »](#)

EXERCISE TIME



MAKE A SIMPLE CALCULATOR

REMOVING A PUNCTUATION FROM A STRING

USEFUL MATERIALS



[PYTHON OFFICIAL TUTORIAL](#)

[W3SCHOOL](#)

[GEEKSFORGEEKS](#)

[EL ZERO WEB SCHOOL](#)

THE END



AND THAT BRINGS US
TO THE END.

I'D LIKE TO THANK
YOU FOR YOUR TIME
AND ATTENTION
TODAY.

By: Youssef M. Khalil



01151751949