Python Functions

What is a function in Python?

- In Python, a function is a group of related statements that performs a specific task.
- Functions help break our program into smaller and modular chunks.
- As our program grows larger and larger, functions make it more organized and manageable.
- Furthermore, it avoids repetition and makes the code reusable.

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()
```

```
Hello from a function
```

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 greet(name):
    print("Hello",name)
    print("Hi")

greet("Ayushi")
```

```
Hello Ayushi
Hi
>
```

Eg2:

```
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

```
Emil Refsnes
Tobias Refsnes
Linus Refsnes
```

Arguments are often shortened to *args* in Python documentations.

Parameters or Arguments?

The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that is sent to the function when it is called.

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")
```

```
Emil Refsnes
```

If you try to call the function with 1 or 3 arguments, you will get an error:

Example

This function expects 2 arguments, but gets only 1:

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

my_function("Emil")
```

```
Traceback (most recent call last):
  File "demo_function_args_error.py", line 4, in <module>
    my_function("Emil")
TypeError: my_function() missing 1 required positional argument: 'lname'
```

The return statement

```
def add_numbers(a,b):
    result=a+b
    print("The sum is",result)

number1=1.2
number2=1.5

add_numbers(number1,number2)
```

```
The sum is 2.7
>
```

```
def add_numbers(a,b):
    result=a+b
    return result

number1=4
number2=1.2
result=add_numbers(number1,number1)
print("The sum is",result)
```

```
The sum is 8
>
```

```
def greet(name):
    print("Hello",name)
    return
    print("Hii")
greet("Ayushi")
```

```
Hello Ayushi
>
```

```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

```
15
25
45
```

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")
```

```
The youngest child is Linus
```

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")
```

```
The youngest child is Linus
```

The phrase *Keyword Arguments* are often shortened to *kwargs* in Python documentations.

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")
```

```
His last name is Refsnes
```

Arbitrary Kword Arguments are often shortened to ***kwargs* in Python documentations.

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")
```

```
I am from Sweden
I am from India
I am from Norway
I am from Brazil
```

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)
```

```
apple
banana
cherry
```

The pass Statement

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

Example

```
def myfunction():
  pass
```

Recursion

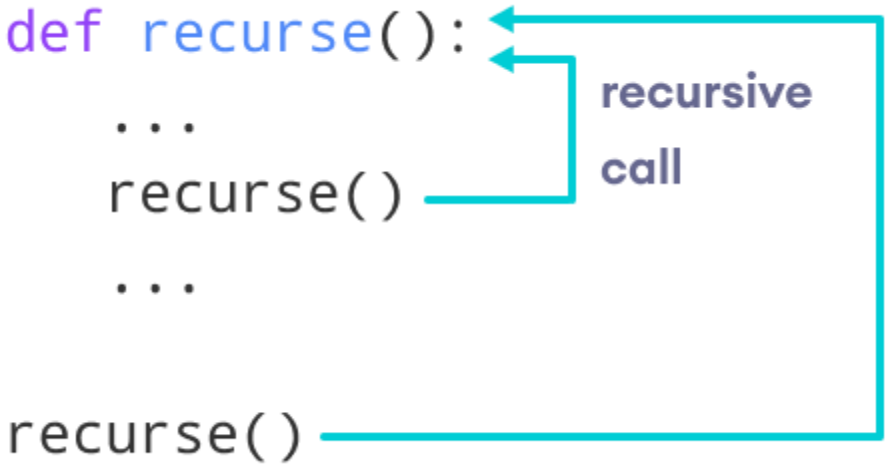
What is recursion?

- Recursion is the process of defining something in terms of itself.

Python Recursive Function

In Python, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

Python also accepts function recursion, which means a defined function can call itself.



Following is an example of a recursive function to find the factorial of an integer.

Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is `1*2*3*4*5*6 = 720`.

Example of a recursive function

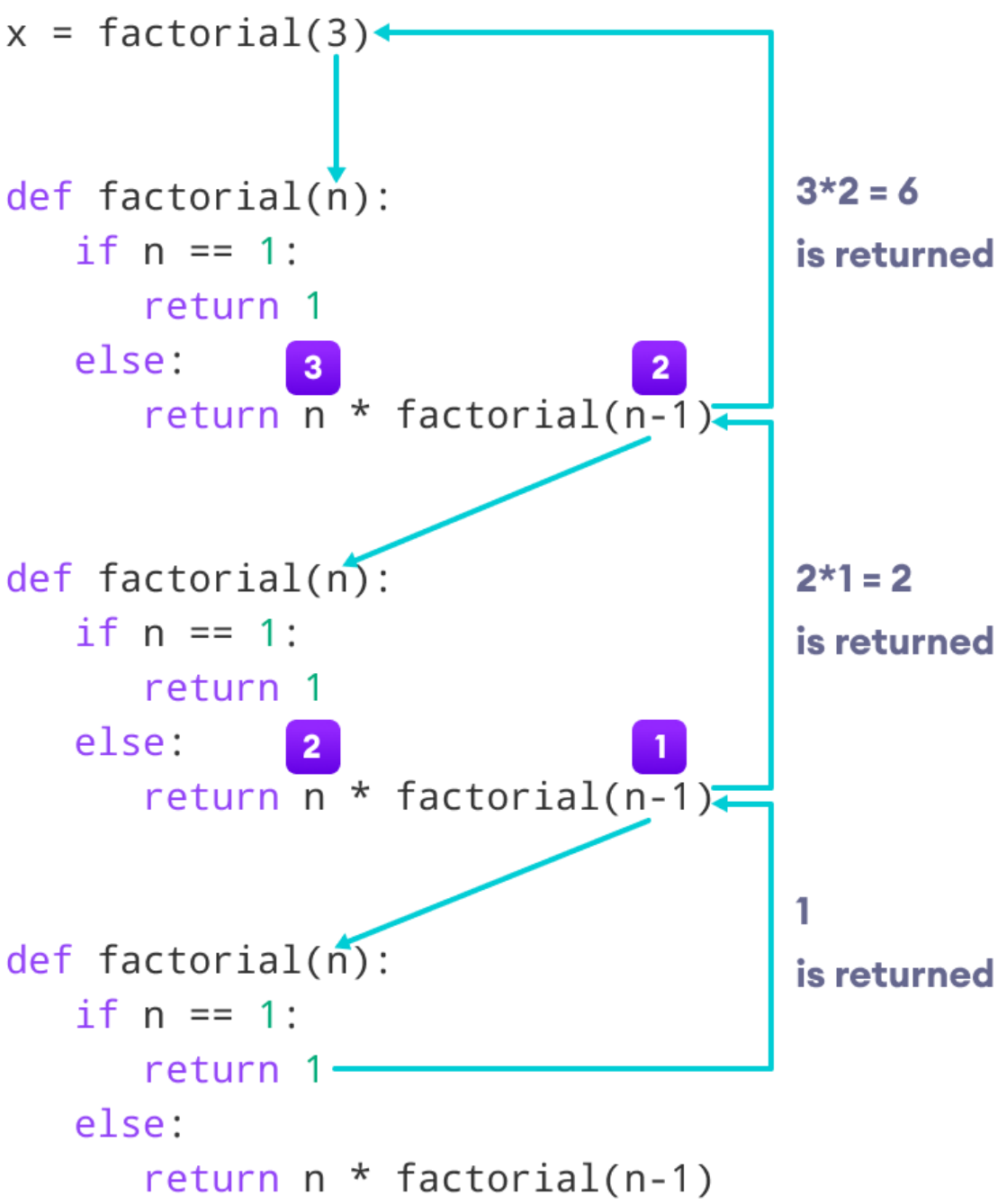
```
def factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""

    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))


num = 3
print("The factorial of", num, "is", factorial(num))
```

Output

```
The factorial of 3 is 6
```


x = factorial(3)
def factorial(n):
if n == 1:
return 1
else:
return n * factorial(n-1)
3
2
3*2 = 6
is returned
def factorial(n):
if n == 1:
return 1
else:
return n * factorial(n-1)
2
1
2*1 = 2
is returned
def factorial(n):
if n == 1:
return 1
else:
return n * factorial(n-1)
1
is returned

Python input() Function

demo_ref_input2.py:

```
x = input("Enter your name:")
print("Hello, " + x)
```

```
C:\Users\My Name>python demo_input2.py
Enter your name:Ayushi
Hello, Ayushi
```

Python Lambda

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

Syntax

```
lambda arguments : expression
```

The expression is executed and the result is returned:

Example

Add 10 to argument `a`, and return the result:

```
x = lambda a : a + 10
print(x(5))
```

```
15
```

Lambda functions can take any number of arguments:

Example

Multiply argument `a` with argument `b` and return the result:

```
x = lambda a, b : a * b
print(x(5, 6))
```

```
30
```

Example

Summarize argument `a`, `b`, and `c` and return the result:

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

```
13
```