# Functions and Recursion

- **Function:** A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.

- **Function definition:** A statement that creates a new function, specifying its name, parameters, and the statements it executes.

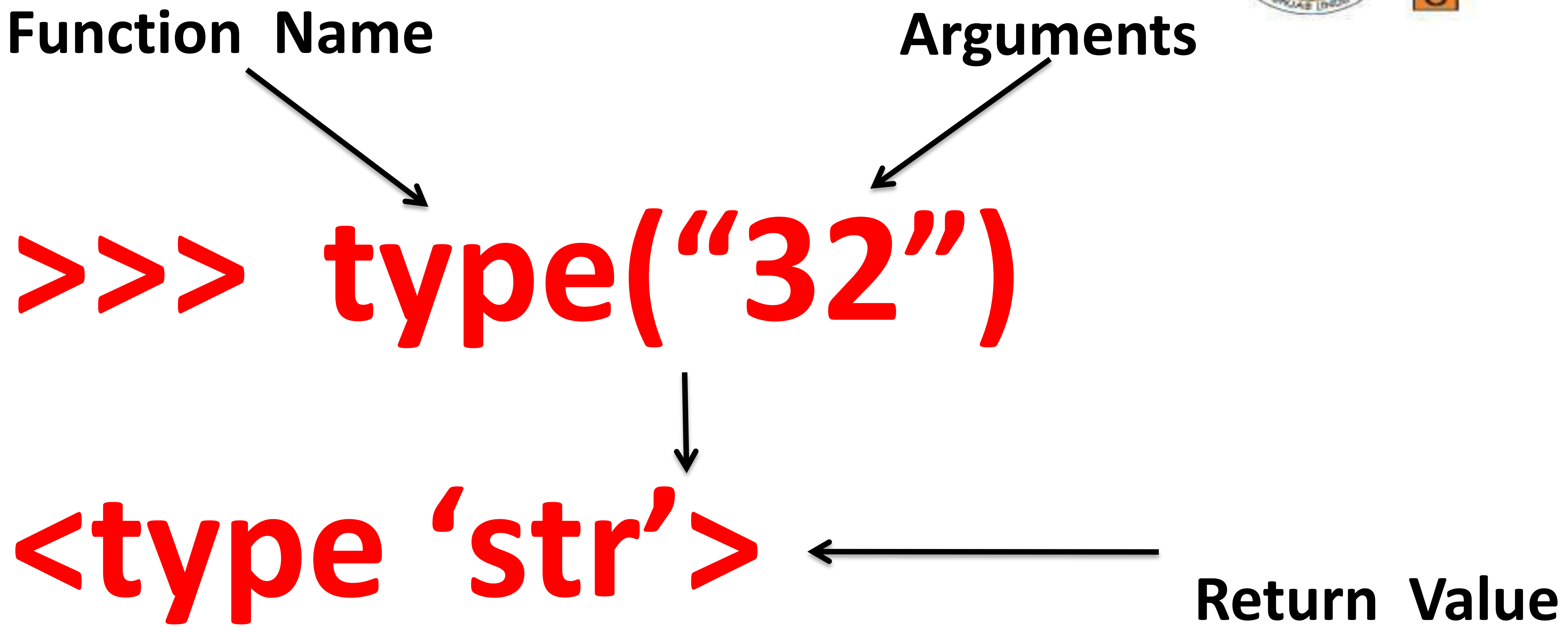
- **Function call:** A statement that call a function definition to perform a specific task.

## Function calls

You have already seen one example of a function call:

```
>>> type("32")
<type 'str'>
```

The name of the function is `type`, and it displays the type of a value or variable.



```
>>> type("32")
<type 'str'>
```

As another example, the `id` function takes a value or a variable and returns an integer that acts as a unique identifier for the value:

```
>>> id(3)
134882108
>>> betty = 3
>>> id(betty)
134882108
```

Every value has an `id`, which is a unique number related to where it is stored in the memory of the computer. The `id` of a variable is the `id` of the value to which it refers.

## Type conversion

Python provides a collection of built-in functions that convert values from one type to another. The `int` function takes any value and converts it to an integer, if possible, or complains otherwise:

```
>>> int("32")
32
>>> int("Hello")
ValueError: invalid literal for int(): Hello
```

## Float to integer conversion

`int` can also convert floating-point values to integers, but remember that it truncates the fractional part:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

## Integer and String Conversion to Float

The `float` function converts integers and strings to floating-point numbers:

```
>>> float(32)
32.0
>>> float("3.14159")
3.14159
```

## Integer and Float Conversion to String

```
>>> str(32)
'32'
>>> str(3.14149)
'3.14149'
```

## Type coercion

- Rules for automatic type conversion is know as **Type coercion.**
- **For example,**
-

```
>>> minute = 59
>>> minute / 60.0
0.983333333333
```

- By making the denominator or numerator a float, we force Python to do floating-point division.

## Math Functions

Python has a math module that provides most of the familiar mathematical functions. A **module** is a file that contains a collection of related functions grouped together.

Before we can use the functions from a module, we have to import them:

```
>>> import math
```

## Calling a Math Function

To call one of the functions, we have to specify the name of the module and the name of the function, separated by a dot, also known as a period. This format is called **dot notation.**

**For Example,**

```
>>> decibel = math.log10 (17.0)
>>> angle = 1.5
>>> height = math.sin(angle)
```

## Composition

Just as with mathematical functions, Python functions can be composed, meaning that you use one expression as part of another. For example, you can use any expression as an argument to a function:

```
>>> x = math.cos(angle + math.pi/2)
```

## Composition

You can also take the result of one function and pass it as an argument to another:

```
>>> x = math.exp(math.log(10.0))
```

This statement finds the log base `e` of 10 and then raises `e` to that power. The result gets assigned to `x`.

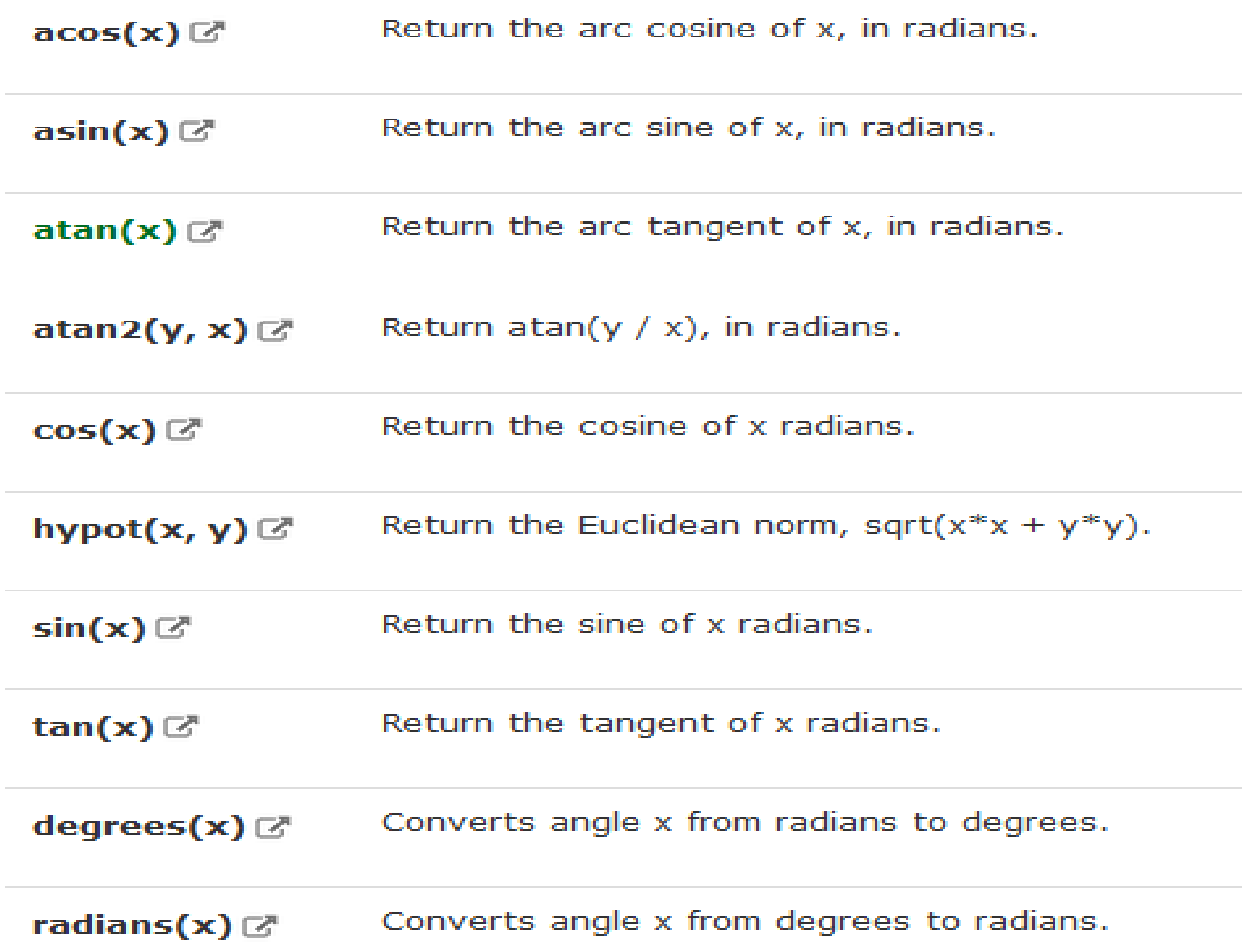
## Mathematical functions

- abs(x) : calculates the absolute value of x
- pow(x , y) : calculates x raise to the power of y.
- Max (x1,x2….xn) : Gives the maximum number out the all the given numbers in parenthesis.
- Min (x1,x2…xn) : Gives the minimum number out the all the given numbers in parenthesis.

## Mathematical Functions stored in Math Package

- exp (x) : exponential
- ceil(x) : ceil value of x
- floor(x) : floor value of x
- log(x) : logarithm with natural base
- sqrt(x) : square root of x

# Trigonometric Functions in Math Module



| | |
|---|---|
| **acos(x)** | Return the arc cosine of x, in radians. |
| **asin(x)** | Return the arc sine of x, in radians. |
| **atan(x)** | Return the arc tangent of x, in radians. |
| **atan2(y, x)** | Return atan(y / x), in radians. |
| **cos(x)** | Return the cosine of x radians. |
| **hypot(x, y)** | Return the Euclidean norm, sqrt(x*x + y*y). |
| **sin(x)** | Return the sine of x radians. |
| **tan(x)** | Return the tangent of x radians. |
| **degrees(x)** | Converts angle x from radians to degrees. |
| **radians(x)** | Converts angle x from degrees to radians. |

## Adding new or user defined functions

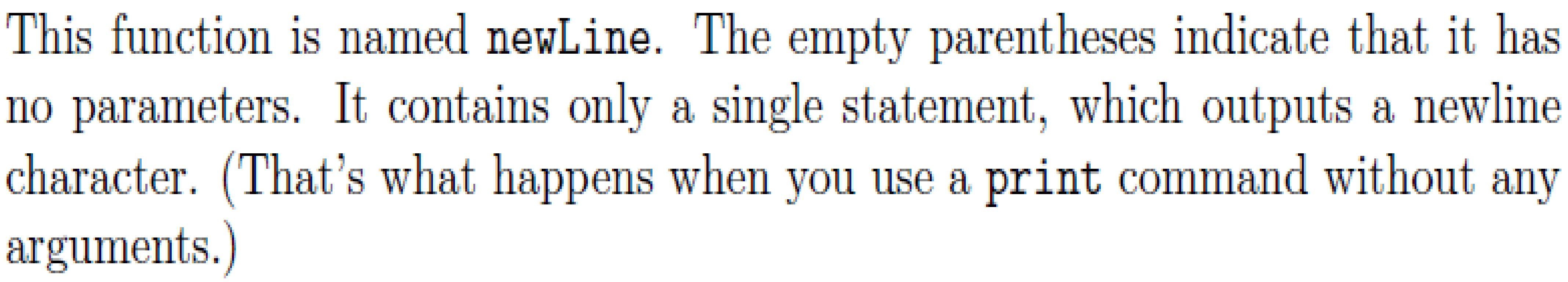
- A function is a named sequence of statements that performs a desired operation. This operation is specified in a **function definition**.

- The syntax for a function definition is:

```
def NAME( LIST OF PARAMETERS ):
   STATEMENTS
```

- **Example,**

```
Def newLine():
    print()
```


This function is named `newLine`. The empty parentheses indicate that it has no parameters. It contains only a single statement, which outputs a newline character. (That's what happens when you use a `print` command without any arguments.)

## Calling user defined functions

The syntax for calling the new function is the same as the syntax for built-in functions:

```
print "First Line."
newLine()
print "Second Line."
```

- **Output of the program is:**

```
First line.

Second line.
```

## Why we need to create functions

- Creating a new function gives you an opportunity to name a group of statements. Functions can simplify a program by hiding a complex computation behind a single command and by using English words in place of arcane code.

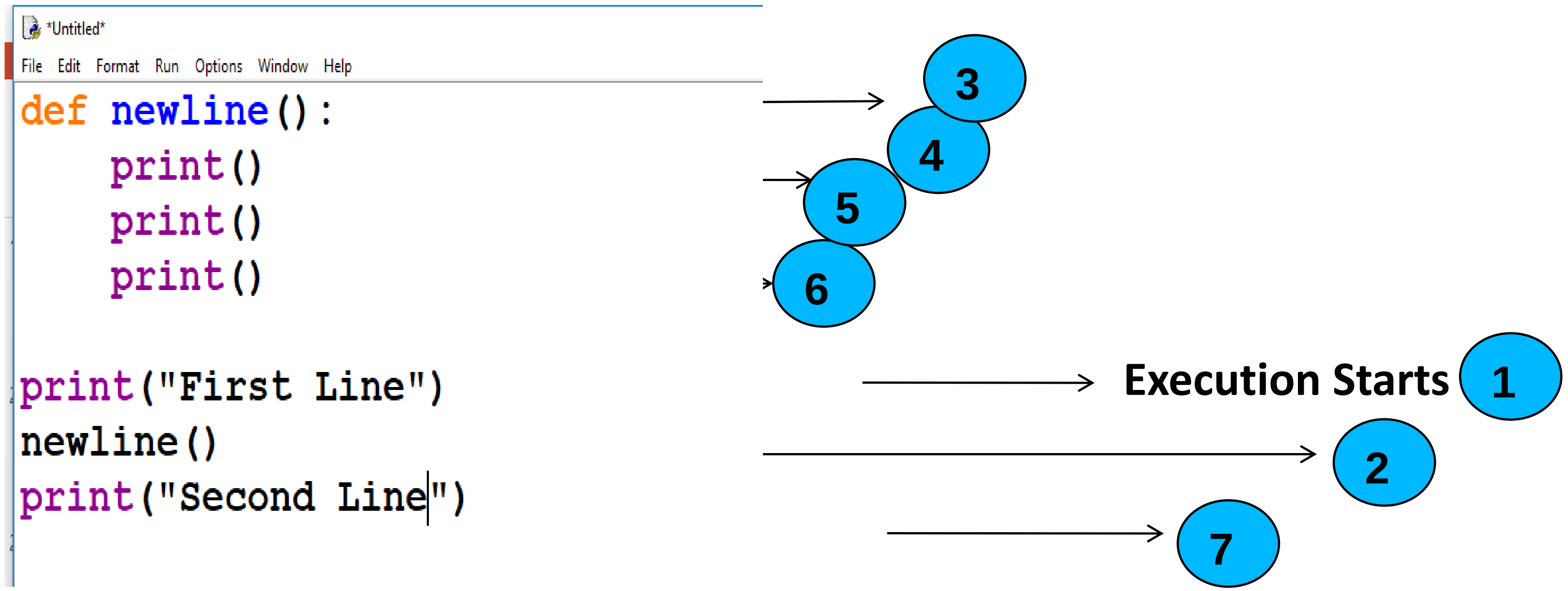
- Creating a new function can make a program smaller by eliminating repetitive code.

## Few points to remember

- Only the function definition generates no output.
- The statements inside the function do not get executed until the function is called.
- You have to create a function before you can execute it. In other words, the function definition has to be executed before the first time it is called.

# Flow of execution

- In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the flow of execution.



```
def newline():
    print()
    print()
    print()

print("First Line")
newline()
print("Second Line")
```

## Parameters and arguments

- **Arguments** are the values that control how the function does its job.
- For example, if you want to find the sine of a number, you have to indicate what the number is. Thus, sin takes a numeric value as an argument.

- Some functions take more than one argument. For example, pow takes two arguments, the base and the exponent. In the function definition, the values that are passed get assigned to variables called as **parameters**.

## EXAMPLE

```
 def mammals( cat, dog):
            animal = cat+dog
            return animal
mice=10
rabbit = 12
mammals(mice , rabbit)
# Here mice and rabbit are arguments
# cat and dog are paramaters
```

```
Def printTwice(twice):
        print(twice,twice)
```

**Function Definition with arguments**

```
>>> printTwice('Spam')
Spam Spam
>>> printTwice(5)
5 5
>>> printTwice(3.14159)
3.14159 3.14159
```

**Function Call with arguments of type String, Integer and float respectively**

## Composition for user defined functions

The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for printTwice:

```
>>> printTwice('Spam'*4)
SpamSpamSpamSpam SpamSpamSpamSpam
>>> printTwice(math.cos(math.pi))
-1.0 -1.0
```

## Variables and parameters are local

- When you create a local variable inside a function, it only exists inside the function, and you cannot use it outside. For example:

```
def catTwice(part1, part2):
    cat = part1 + part2
    print (cat)

>>> a= “Python”
>>> b=“Class”
>>> catTwice(a,b)
```

When cat Twice terminates, the variable cat is destroyed. If we try to print it,

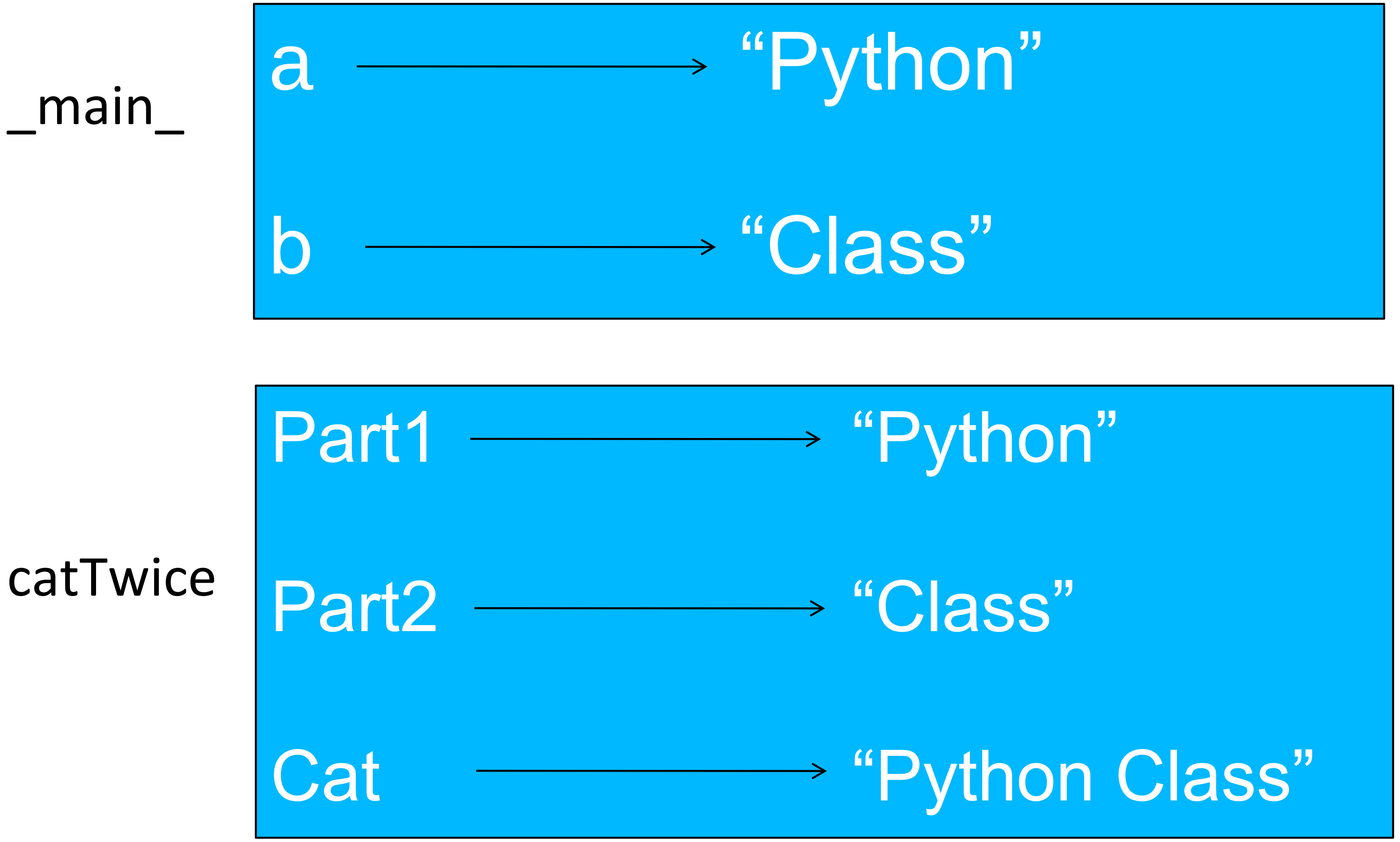
we get an error:

```
>>> print( cat)
NameError: cat
```

- Parameters are also local. For example, outside the function **catTwice(part1, part2),** there is no such thing as **part1** and **part2**

## Stack Diagram for functions

- To keep track of which variables can be used where, it is sometimes useful to draw a stack diagram.
- Stack diagram is used to represent the state of a program during a function call.
- Each function is represented by a frame. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it.


_main_
a
“Python”
b
“Class”
catTwice
Part1
“Python”
Part2
“Class”
Cat
“Python Class”

## Functions with results

```
def  sum (x,y):
       return x+y


>>> a=sum(8,9)
>>> print(a)
17
```

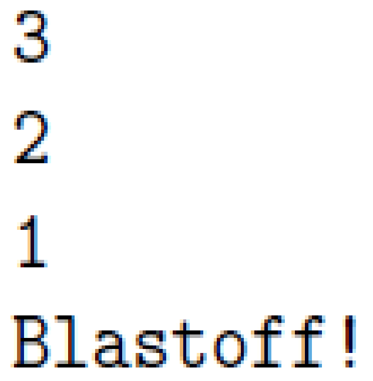
## Recursion

- It is legal for one function to call another, and you have seen several examples of that.
- But it is also legal for a function to call itself.
- **For example,**

```
def countdown(n):
    if n==0:
        print("blastoff")
    else:
        print(n)
        countdown(n-1)
```

```
>>> countdown(3)
```

**OUTPUT of this function will be :**



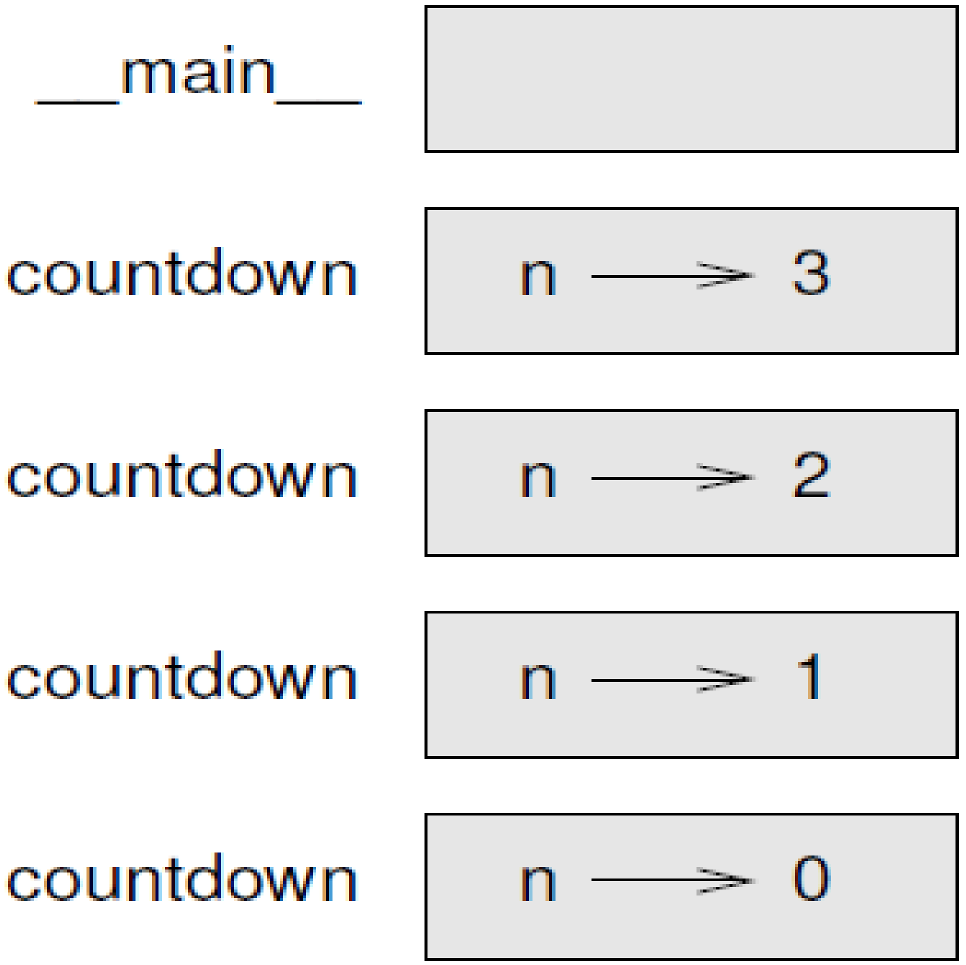
```
3
2
1
Blastoff!
```

## Stack diagrams for recursive functions

- Earlier we used a stack diagram to represent the state of a program during a function call. The same kind of diagram can help interpret a recursive function.

- Every time a function gets called, Python creates a new function frame, which contains the function's local variables and parameters. For a recursive function, there might be more than one frame on the stack at the same time.

- Following figure shows a stack diagram for function **countdown (n)** called with n = 3:



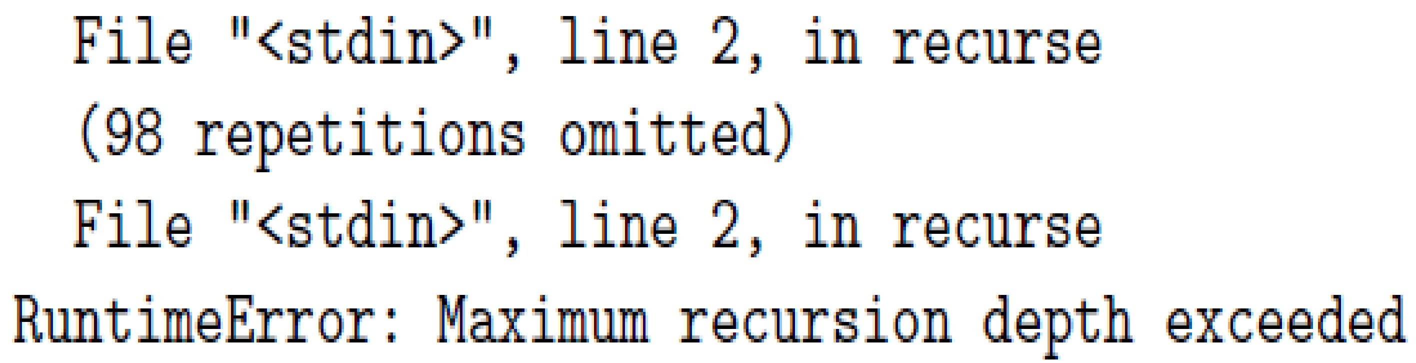
## Infinite recursion

- If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as infinite recursion, and it is generally not considered a good idea. Here is a minimal program with an infinite recursion:

```
def  recurse () :
        recurse()
```

**Function is calling itself without any condition to terminate The recursive call**

- In most programming environments, a program with infinite recursion does not really run forever. Python reports an error message when the maximum recursion depth is reached:



```
  File "<stdin>", line 2, in recurse
  (98 repetitions omitted)
  File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

## EXCERCISE

1. Write a Python function to find the Max of three numbers.
2. Write a Python function to sum all the numbers in a list.
3. Write a Python function to multiply all the numbers in a list.
4. Write a Python function to reverse a string.
5. Write a Python function to calculate the factorial of a number (non-negative integer). The function accept the number as an argument.

6. Write a Python function to check whether a number is in a given range.
7. Write a Python function that takes a number as a parameter and check the number is prime or not.
8. Write a Python function that checks whether a passed number is palindrome or not.

9. Write a Python function to check whether a number is perfect or not. According to Wikipedia : In number theory, a perfect number is a positive integer that is equal to the sum of its proper positive divisors, that is, the sum of its positive divisors excluding the number itself (also known as its aliquot sum). Equivalently, a perfect number is a number that is half the sum of all of its positive divisors (including itself).

*Example* : The first perfect number is 6, because 1, 2, and 3 are its proper positive divisors, and 1 + 2 + 3 = 6. Equivalently, the number 6 is equal to half the sum of all its positive divisors: ( 1 + 2 + 3 + 6 ) / 2 = 6. The next perfect number is 28 = 1 + 2 + 4 + 7 + 14. This is followed by the perfect numbers 496 and 8128.