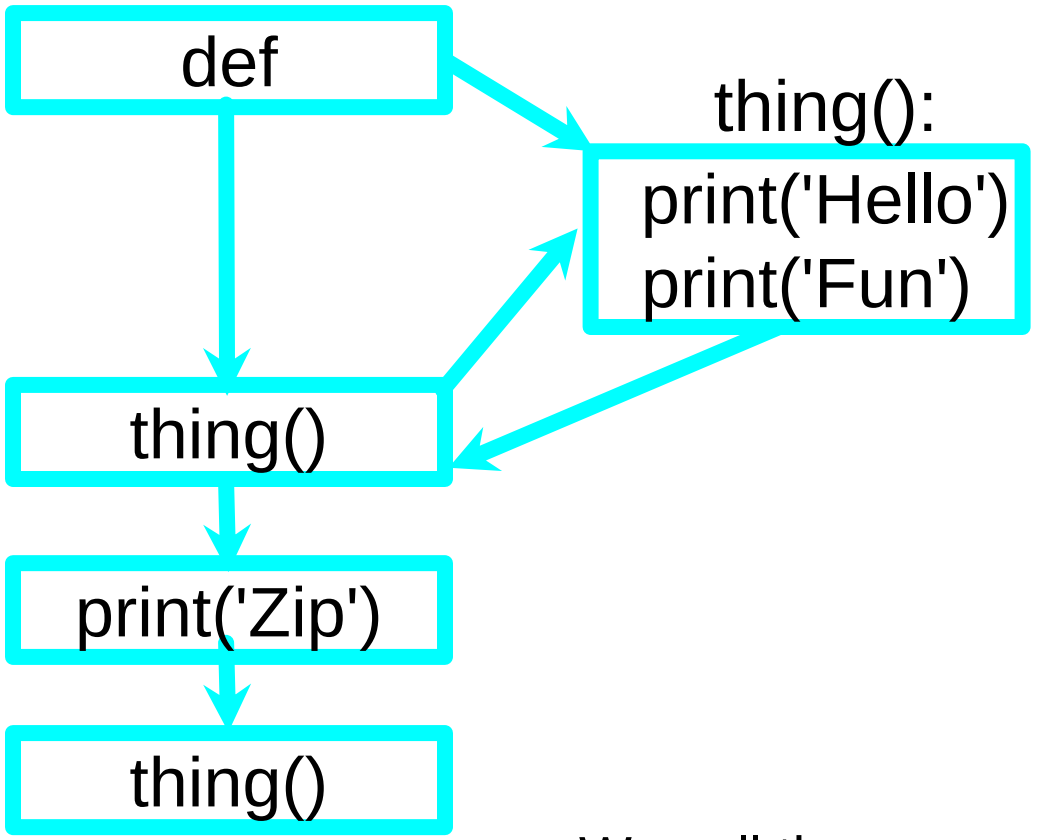
CS1011 Computer Programming in Python

Lecture 4:Functions

Objectives

1. **Defining Functions & Calling Functions**
    1. Built-in vs user-defined functions
    2. Function syntax & examples
2. **Function Parameters & Arguments**
    1. No argument, single argument, multiple arguments
    2. Variable-length arguments (*args, **kwargs)
3. **Return Values**
    1. Fruitful (with return) vs Void (no return) functions
4. **Scope of Variables**
    1. Local variables
    2. Global variables
    3. Modifying global variables
5. **Recursion**
    1. Concept of recursion
    2. Example: factorial

Stored (and reused) Steps



Program:

```
def thing():
    print('Hello')
    print('Fun')

thing()
print('Zip')
thing()
```

Output:

```
Hello
Fun
Zip
Hello
Fun
```

We call these reusable pieces of code “functions”

Python Functions

There are two kinds of functions in Python.

- Built-in functions that are provided as part of Python - print(), input(), type(), float(), int() ...

- Functions that we define ourselves and then use

We treat function names as “new” reserved words
(i.e., we avoid them as variable names)

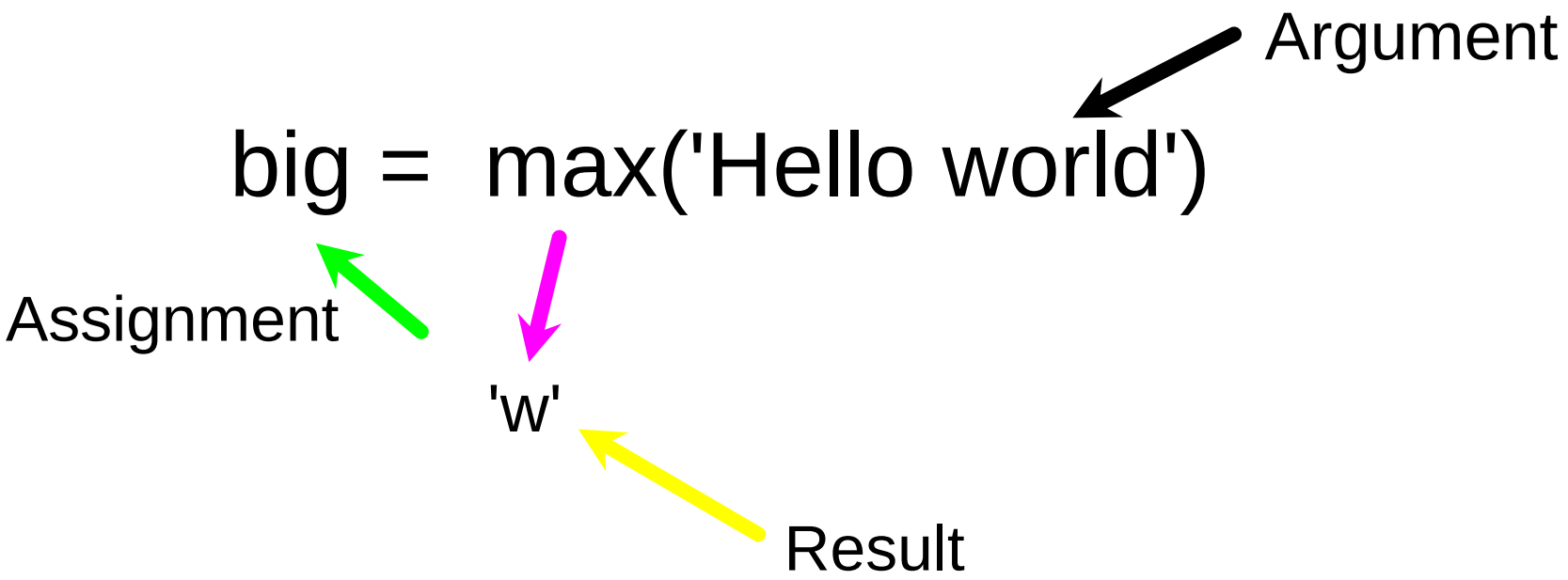
Function Definition

In Python a function is some reusable code that takes arguments(s) as input, does some computation, and then returns a result or results

We define a function using the def reserved word

We call/invoke the function by using the function name, parentheses, and arguments in an expression

Max Function



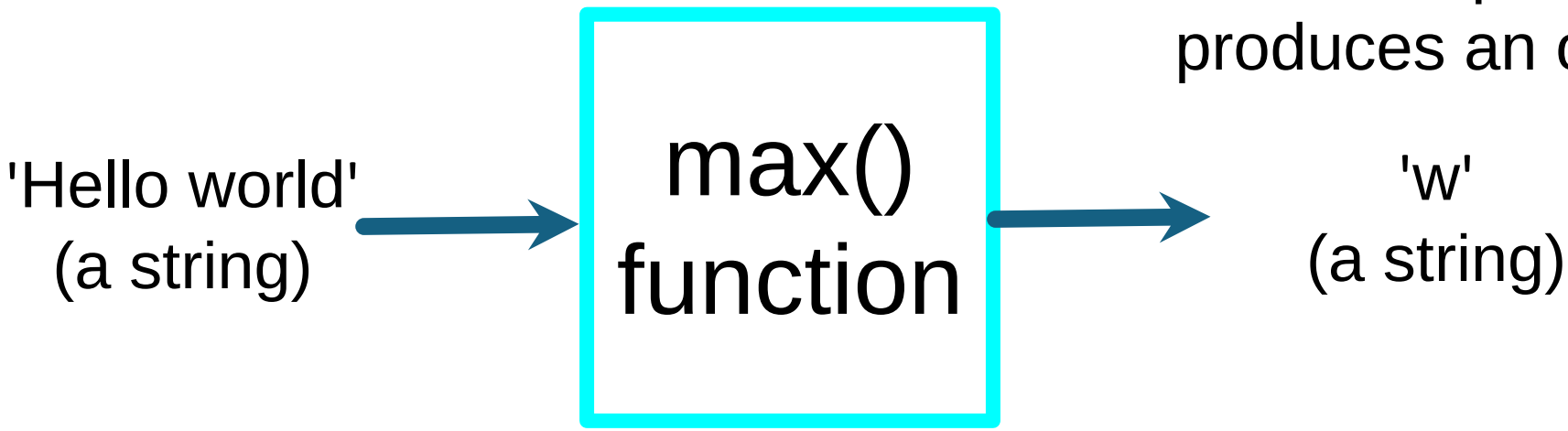

```
>>> big = max('Hello world')
>>> print(big)
w
>>> tiny = min('Hello world')
>>> print(tiny)

>>>
```

Max Function

```
>>> big = max('Hello world')
>>> print(big)
w
```

A function is some stored code that we use. A function takes some input and produces an output.



Guido wrote this code

Type Conversions

1. Implicit Conversion

When you put an integer and floating point in an expression, the integer is implicitly converted to a float.

```
print(99 / 100) # 0.99
```

Here, 99 is converted into 99.0 before division.

Type Conversions

2. Explicit Conversion

- You can control this with the built-in functions int() and float()

```
i = 42
print(type(i))       # <class 'int'>

f = float(i)
print(f)             # 42.0
print(type(f))       # <class 'float'>
```

String Conversions

"123" is stored as a **string**, not a number.

```
1  sval = "123"
2  print(type(sval))   # <class 'str'>
```

Output

```
<class 'str'>
```

❌ Error: you cannot add a **string** and an **integer**.

main.py

```
1  sval = '123'
2  type(sval)
3  print(sval + 1)
```

Output Clear

```
ERROR!
Traceback (most recent call last):
  File "<main.py>", line 3, in <module>
TypeError: can only concatenate str (not "int") to str
```

String Conversions

You can use int() to convert numeric strings into integers, and float() to convert numeric strings into floating-point numbers.
Here, "123" (a string) is converted to 123 (an integer) with int().

```
1  sval = '123'
2  type(sval)
3  ival = int(sval)
4  type(ival)
5  print(ival + 1)
```

Output

```
124
```

You will get an error if the string does not contain numeric characters

```
nsv = 'hello bob'
niv = int(nsv)
```

```
ERROR!
Traceback (most recent call last):
  File "<main.py>", line 3, in <module>
ValueError: invalid literal for int() with base 10:
    'hello bob'
```

❌ Error: non-numeric strings (like "hello bob") **cannot be converted** to integers.

Building our Own Functions

We create a new function using the def keyword followed by optional parameters in parentheses

We indent the body of the function

This defines the function but does not execute the body of the function

```
2 def print_greeting():
3     print("Good morning, students!")
4     print("Welcome to learning Python.")
5
```

Definitions and Uses

Once we have defined a function, we can call (or invoke) it as many times as we like. This is the store and reuse pattern

```
1  x = 5
2  print("Hello")
3
4  def print_greeting():
5      print("Good morning, students!")
6      print("Welcome to learning Python.")
7
8  print("Yo")
9  x = x + 2
10 print(x)
11
12 # Call the function
13 print_greeting()
```

Output

```
Hello
Yo
7
Good morning, students!
Welcome to learning Python.
```

Arguments

An argument is a value we pass into the function as its input when we call the function

We use arguments so we can direct the function to do different kinds of work when we call it at different times

We put the arguments in parentheses after the name of the function

```
big = max('Hello world')
```

Argument

Parameters

A parameter is a variable which we use in the function definition. It is a “handle” that allows the code in the function to access the arguments for a particular function invocation.

```
def greet(lang):
    if lang == 'es':
        print('Hola')
    elif lang == 'fr':
        print('Bonjour')
    else:
        print('Hello')


# Function calls
greet('en')   # Output: Hello
greet('es')   # Output: Hola
greet('fr')   # Output: Bonjour
```

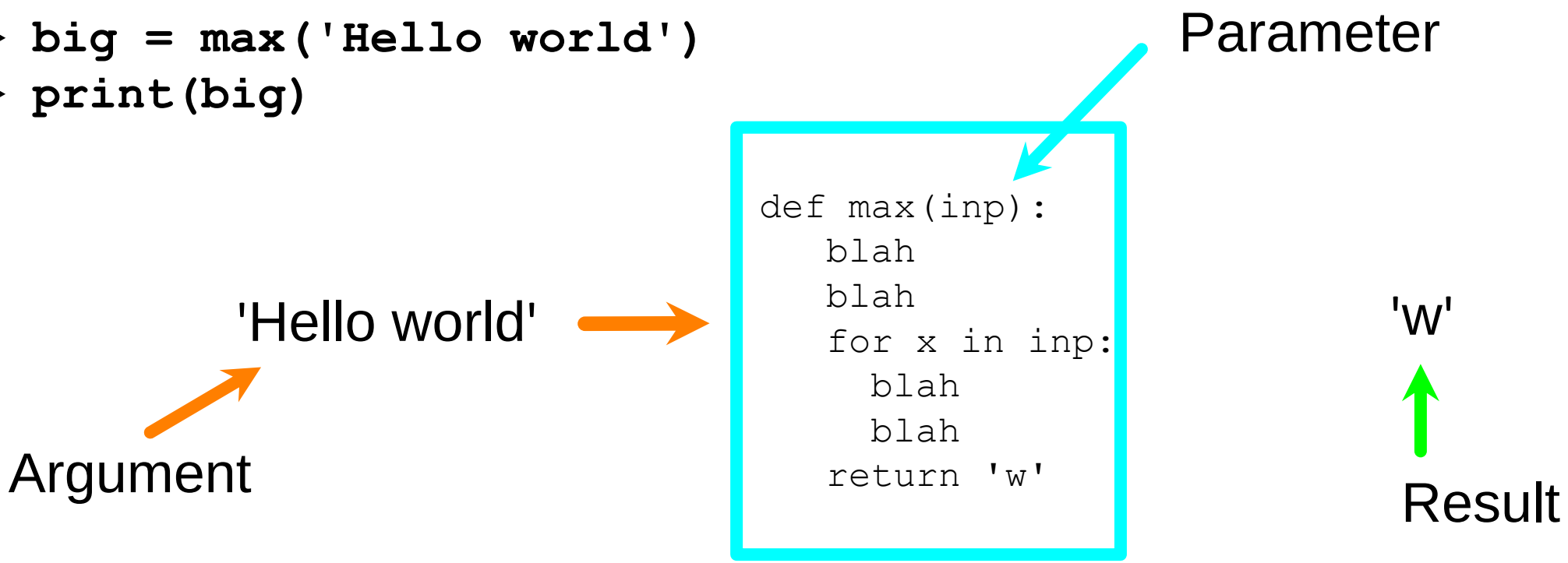
```
Output

Hello
Hola
Bonjour

=== Code Execution Successful ===
```

Arguments, Parameters, and Results

```
>>> big = max('Hello world')
>>> print(big)
w
```



Parameters / Arguments

We can define more than one parameter in the function definition

We simply add more arguments when we call the function

We match the number and order of arguments and parameters

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print(x)
```

```
Output

8
```

Passing No Argument to a Function

- we can pass no arguments at all to a function

```
1  # no argument is passed
2
3  # function definition
4  def displayMessage():
5      print("Learning Functions in Python")
6
7  # function call
8  displayMessage()
9  # Output: Learning Functions in Python
0
```

Output

```
Learning Functions in Python

=== Code Execution Successful ===
```

Passing a Single Argument to a Function

- we can pass single arguments to a function

```
1  # single argument is passed
2
3  # function definition
4  def displayMessage(msg):
5      print("Learning " + msg + " in Python")
6
7  # function call
8  displayMessage("Functions")
9  # Output: Learning Functions in Python
```

```
Output

Learning Functions in Python

=== Code Execution Successful ===
```

Passing Multiple Arguments to a Function

- We can pass multiple arguments to a python function by predetermining the formal parameters in the function definition.

```
1 def displayMessage(word1, word2, word3):
2     print(word1, word2, word3)
3
4 displayMessage("Learning", "Python", "Functions")
```

```
Output

Learning Python Functions

=== Code Execution Successful ===
```

Passing Variable Number of Arguments (*args)

- We can pass multiple arguments to a python function without predetermining the formal parameters
- def functionName(*argument)
- Typically, this syntax is used to avoid the code failing when we don't know how many arguments will be sent to the function.
- The *argument is saved as tuple

```
1  # variable number of non keyword arguments passed
2
3  # function definition
4  def calculateTotalSum(*arguments):
5      totalSum = 0
6      for number in arguments:
7          totalSum += number
8      print(totalSum)
9
10 # function call
11 calculateTotalSum(5, 4, 3, 2, 1)
```

```
Output

15

=== Code Execution Successful ===
```

Passing Variable Number of Keyword Arguments (**kwargs)

- We can pass multiple keyword arguments to a python function without predetermining the formal parameters using the below syntax:
- def functionName(**argument)
- The ** symbol is used before an argument to pass a keyword argument dictionary to a function, this syntax used to successfully run the code when we don't know how many keyword arguments will be sent to the function.

Passing Variable Number of Keyword Arguments (**kwargs)

```
1 def show_kwargs(**kwargs):
2     print(kwargs)
3
4 show_kwargs(name="Ali", age=25)
```

Output

```
{'name': 'Ali', 'age': 25}

=== Code Execution Successful ===
```

Types of Arguments in Python

```python
# single argument, non-keyword arguments,
# and keyword arguments are passed

# function definition
def displayArguments(arg1, *args, **kwargs):

    # displaying predetermined argument
    print(arg1)

    # displaying non-keyword arguments
    for arg in args:
        print(arg)

    # displaying keyword arguments
    for key, value in kwargs.items():
        print(key, ":", value)

# function call
displayArguments("Learning", "Functions", "in", "Python",
                 level="Beginner", topic="Functions")

```

Output

```
Learning
Functions
in
Python
level : Beginner
topic : Functions

=== Code Execution Successful ===
```

Example

```
def mult(arg1, *arg2, **arg3):
  print('the first arguemt is: ', arg1, 'and it is saved with type:', type(arg1))
  print('the arguemts in between are: ', arg2, 'and they are saved with type:', type(arg2))
  print('the last two arguemts are: ', arg3, 'and they are saved with type:', type(arg3))


mult(3, 4,'k','N', a=1, b=2)
```

Output

```
the first arguemt is:  3 and it is saved with type: <class 'int'>
the arguemts in between are:  (4, 'k', 'N') and they are saved with type: <class 'tuple'>
the last two arguemts are:  {'a': 1, 'b': 2} and they are saved with type: <class 'dict'>
```

Return Values

Often a function will take its arguments, do some computation, and return a value to be used as the value of the function call in the calling expression. The return keyword is used for this.

```
1 def greet():
2     return "Hello"
3
4 print(greet(), "Glenn")
5 print(greet(), "Sally")
```

```
Output
Hello Glenn
Hello Sally
```

Return Values

A “fruitful” function is one that produces a result (or return value)

The return statement ends the function execution and “sends back” the result of the function

```
1  def greet(lang):
2      if lang == 'es':
3          return 'Hola'
4      elif lang == 'fr':
5          return 'Bonjour'
6      else:
7          return 'Hello'
8
9
10 # Function calls
11 print(greet('en'), 'Glenn')     # Hello Glenn
12 print(greet('es'), 'Sally')     # Hola Sally
13 print(greet('fr'), 'Michael')   # Bonjour Michael
```

```
Output

Hello Glenn
Hola Sally
Bonjour Michael
```

Void (non-fruitful) Functions

When a function does not return a value, we call it a “void” function

Functions that return values are “fruitful” functions

Void functions are “not fruitful”

To function or not to function

Organize your code into “paragraphs” - capture a complete thought and “name it”

Don’t repeat yourself - make it work once and then reuse it

If something gets too long or complex, break it up into logical chunks and put those chunks in functions

Make a library of common stuff that you do over and over - perhaps share this with your friends...

Namespaces and scopes

Namespace

A namespace (sometimes also called a context) is a naming system for making names unique to avoid ambiguity.

- Naming people: firstname surname [birthday][birthplace]
- Naming websites: subdomain.domain.top-level-domain

Scope

The scope of a name is the area of a program where this name can be unambiguously used, for example inside of a function.

Namespaces and scopes

To associate a name, with a particular namespace, Python uses the location of the assignment of such name

In other words, the place where you assign a name in your source determines the namespace it will live in, and hence its scope of visibility.

Local variables

By default, all names assigned inside a function are associated with that function's namespace (local namespace)

```
def func():
    x = 88
    print("Inside", x)

func()
print("Outside", x)
```

Local variables

By default, all names assigned inside a function are associated with that function's namespace (local namespace)

```
def func():
    x = 88
    print("Inside", x)

func()
print("Outside", x)
```

- Names assigned inside a def can only be seen by the code within that def.
- x is called a local variable

```
Inside 88
Traceback (most recent call last):
  File "lecture.py", line 6, in <module>
    print("Outside", x)
NameError:   name 'x' is not defined
```

Global variables

Names defined outside functions are associated with the global namespace.

```
# Var defined before the
# function and the call
x = 88
def func():
    print("Inside", x)

func()
print("Outside", x)
```

Global variables

Names defined outside functions are associated with the global namespace.

```
# Var defined before the
# function and the call
x = 88
def func():
    print("Inside", x)

func()
print("Outside", x)
```

```
Inside 88
Outside 88
```

- Names assigned outside a `def` can be seen by functions, *provided that they are defined before the function is called.*
- `x` is called a global variable

Global variables

Names defined outside functions are associated with the global namespace.

```
def func():
    print("Inside", x)

# Var defined before the call
x = 88
func()
print("Outside", x)
```

Global variables

Names defined outside functions are associated with the global namespace.

```
def func():
    print("Inside", x)

# Var defined before the call
x = 88
func()
print("Outside", x)
```

```
Inside 88
Outside 88
```

- Names assigned outside a `def` can be seen by functions, *provided that they are defined before the function is called.*
- `x` is called a global variable

Global variables

```
def func():
    print("Inside", x)

func()
# Var defined after the call
x = 88
print("Outside", x)
```

Global variables

```
def func():
    print("Inside", x)

func()
# Var defined after the call
x = 88
print("Outside", x)
```

```
Inside 88
Traceback (most recent call last):
  File "lecture.py", line 2, in func
    print("Inside", x)
NameError:   name 'x' is not defined
```

- Names assigned outside a `def` can be seen by functions, *provided that they are defined before the function is called.*
- `x` was not defined before the call

Local and global variables

If a variable exists in both the local and global namespace, the copies are distinct.

```
x = 99

def func():
    x = 88
    print("Inside", x)

func()
print("Outside", x)
```

Local and global variables

If a variable exists in both the local and global namespace, the copies are distinct.

```
x = 99

def func():
    x = 88
    print("Inside", x)

func()
print("Outside", x)
```

```
Inside 88
Outside 99
```

- Inside the function, the local namespace for x is used.
- Outside the function, the global namespace for x is used.

Global variables

- If you want to **modify** a global variable inside a function, you must declare it with the keyword global.

```
1  x = 88
2
3  def func():
4      global x        # declare x as global
5      x = x + 1       # modifies the global variable
6      print("Inside", x)
7
8  func()
9  print("Outside", x)
10
```

Output

```
Inside 89
Outside 89

=== Code Execution Successful ===
```

Recursive functions

- Recursive function is a function that calls itself as part of its execution.
- Recursive functions are used to solve problems that can be broken down into smaller instances of the same problem.
- Each recursive call works on a smaller piece of the problem until it reaches a base case where a direct solution can be obtained without further recursion.
- Recursive functions can be a powerful tool for solving certain types of problems, but they should be designed carefully to avoid **infinite recursion and excessive function calls.**

Recursive function example

- Suppose we want to calculate the factorial value of an integer [n! = n * (n-1)!]

```
def fact_loop(n):
  fact = 1
  for i in range(1, n+1):
    fact = fact*i
  return fact

fact_loop(5)
```

```
120
```

```
def fact(n):
  if n==0:
    return 1
  return n*fact(n-1)

fact(5)
```

```
120
```