

# Introduction to Python

## **Functions**

# Topics

- 1) Functions
- 2) Function Inputs vs Outputs
- 3) Function Arguments
  - a) Positional Arguments
  - b) Keyword Arguments
  - c) Default Values
- 4) Flow of Program
- 5) Scope of a variable
- 6) Template for programs

# Example

Consider the following code which asks the user to enter a number and prints out the absolute value of the number. This problem was a lab assignment in the last lecture.

```
x = int(input('Enter an integer: '))  
if x >= 0:  
    print("The absolute value of", x, "is", x)  
else:  
    print("The absolute value of", x, "is", -x)
```

# Example

Now what if the program asks the user for two numbers and then compute their absolute values? What do you think of the following code?

```
x = int(input('Enter an integer: '))  
if x >= 0:  
    print("The absolute value of", x, "is", x)  
else:  
    print("The absolute value of", x, "is", -x)
```

Note the redundancy!  
We like to reuse code  
without rewriting or  
copying/pasting  
code!

```
x = int(input('Enter an integer: '))  
if x >= 0:  
    print("The absolute value of", x, "is", x)  
else:  
    print("The absolute value of", x, "is", -x)
```

# Functions

One way to organize Python code and to make it more readable and reusable is to factor out useful pieces into reusable *functions*.

A **function** is a **named** group of programming instructions that accomplish a specific task. If we want to perform the task, we simply "call" the function **by its name**. A function may be called as many times as we wish to redo the task.

The "30 seconds" button on the microwave is an example of a function. If we press it (call it by its name), it will run the microwave 30 seconds. Later, if we want to heat something else, we can press it again to run the microwave another 30 seconds.

In other programming languages, functions are also called **procedures** or **methods**.

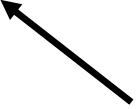
# Example

We like to take the redundant code below and convert it to a function.

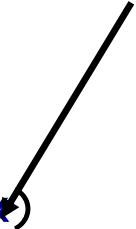
```
x = int(input('Enter an integer: '))  
if x >= 0:  
    print("The absolute value of", x, "is", x)  
else:  
    print("The absolute value of", x, "is", -x)
```

```
x = int(input('Enter an integer: '))  
if x >= 0:  
    print("The absolute value of", x, "is", x)  
else:  
    print("The absolute value of", x, "is", -x)
```

Let's factor out this piece of code, convert it into a function by giving it a name!



Then we can call it repeatedly if we wish to run the code.



# Functions

A **function** or **procedure** is a group of code that has a name and can be called using parentheses.

A function may have **parameters or input variables** to the function. Parameters are input variables that provide information to the function to accomplish its task.

In Python, a function is defined using the *def* statement.

```
def function_name(parameters):  
    block of code
```

# Absolute Value

We like to take the redundant code below and convert it to a function called `absolute()`.

```
def absolute(x):  
    if x >= 0:  
        print("The absolute value of", x, "is", x)  
    else:  
        print("The absolute value of", x, "is", -x)
```

We placed the code into a function named `absolute()`.

This block of code is called the function definition.

```
# several calls to abs()  
absolute(-10)    # The absolute value of -10 is 10  
absolute(5)      # The absolute value of 5 is 5
```

The function definition must precede any function calls.

Now we can reuse this code by calling on `absolute()` with different inputs!



# Absolute Value

```
def absolute(x):  
    if x >= 0:  
        print("The absolute value of", x, "is", x)  
    else:  
        print("The absolute value of", x, "is", -x)
```

absolute(-10)

absolute(5)

The first time absolute() is called, input x variable has the value of -10

Once this function call is done executing. This value of x is released from memory.

# Absolute Value

```
def absolute(x):  
    if x >= 0:  
        print("The absolute value of", x, "is", x)  
    else:  
        print("The absolute value of", x, "is", -x)
```

absolute(-10)

absolute(5)

The second time absolute() is called, a new variable x is created with the value 5.

Once this function call is done executing. This value of x is again released from memory.

# Function Outputs

The previous example prints out a message as part of its output. But what if another programmer who wishes to use our function does not want that message printed? Or if another programmer simply wants the output to be used in another calculation?

We typically want functions to **output** or **return** some answer. The answer can then be printed in a message or used in a different calculation.

```
def absolute(x):
```

```
    if x >= 0:
```


```
        return x
```

```
    else:
```

```
        return -x
```

the function returns or outputs 10  
which is stored in the expression  
*absolute*(-10)

```
print("The absolute value of -10 is", absolute(-10))
```



# Function Outputs

If a function returns a value, the function call expression represents the returned value!

For example, below, the expression `absolute(-10)` is equal to the returned value of 10.

```
def absolute(x):
```

```
    if x >= 0:
```

```
        return x
```

```
    else:
```

```
        return -x
```

the function returns or outputs 10  
which is stored in the expression  
`absolute(-10)`

```
print("The absolute value of -10 is", absolute(-10))
```



This function notation is perfectly consistent with the math notation used in algebra.

If  $f(x) = 3x$ , then the expression  $f(5)$  is equal to 15 and the expression  $f(10)$  is equal to 30.

The expression `absolute(-10)` is equal to 10.

# Function Outputs

The output or returned value can be used in another calculation.

```
def absolute(x):
```

```
    if x >= 0:
```

```
        return x
```

```
    else:
```

```
        return -x
```

Here the returned value is used in another calculation.



```
print("The absolute value of -10 is", absolute(-10))
```

```
x = absolute(-5) + 3
```

```
print(x)    # 8
```


# Function Outputs

```
def absolute(x):  
    if x >= 0:  
        return x  
    else:  
        return -x
```

The important takeaway here is:  
Functions should NOT print the  
answer. It should RETURN the  
answer!

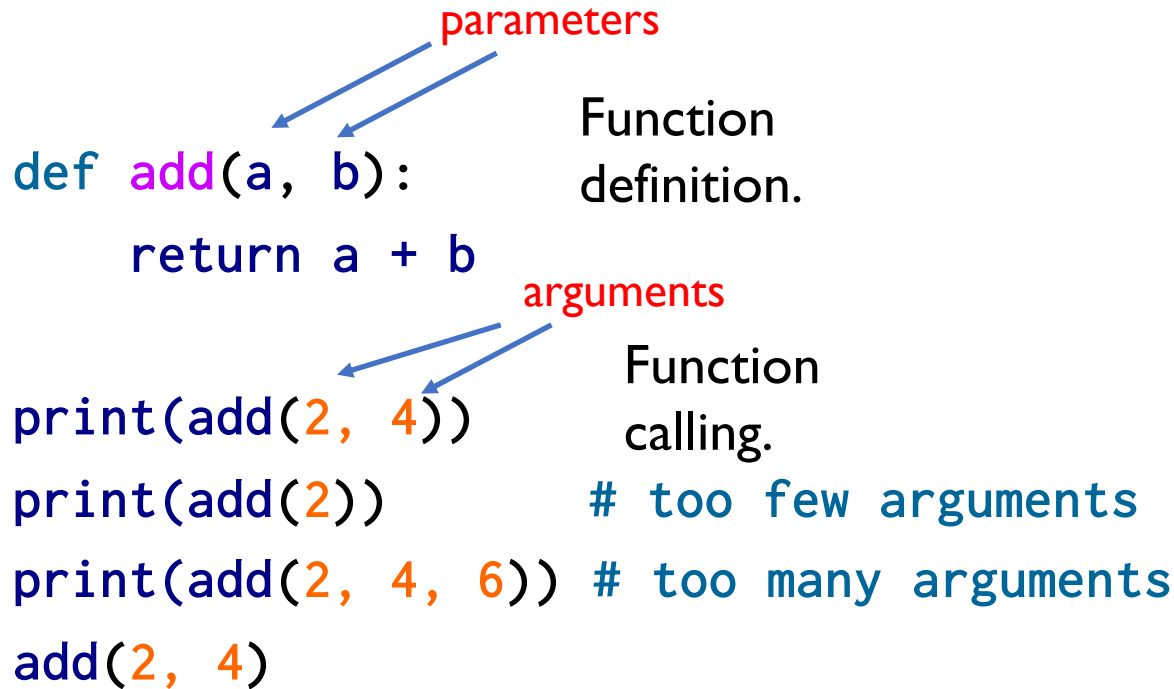
Printing should be done outside the  
function. Print the returned value.

```
print("The absolute value of -10 is", absolute(-10))  
x = absolute(-5) + 3  
print(x)    # 8
```



# Functions

The **arguments** of a function are the inputs of the function. By default, arguments are positional.



The diagram illustrates the relationship between function definitions and calls. It features two main sections: 'Function definition' and 'Function calling'. The 'Function definition' section shows the code `def add(a, b):` followed by `return a + b`. The word 'parameters' is written in red above the parameters `a` and `b`, with two blue arrows pointing to them. The 'Function calling' section shows three lines of code: `print(add(2, 4))`, `print(add(2))`, and `print(add(2, 4, 6))`, followed by `add(2, 4)`. The word 'arguments' is written in red above the arguments in the first call, with two blue arrows pointing to `2` and `4`. The second and third calls are annotated with '# too few arguments' and '# too many arguments' respectively, indicating errors. The final line `add(2, 4)` is also shown.

```
def add(a, b):  
    return a + b  
  
print(add(2, 4))  
print(add(2))  
print(add(2, 4, 6))  
add(2, 4)
```

parameters

Function definition.

arguments

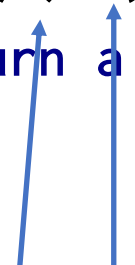
Function calling.

# too few arguments

# too many arguments

# Functions Arguments (input)

```
def add(a, b):  
    return a + b  
  
a = add(2, 4)  
print(a)           # 6
```

Two blue arrows originate from the arguments '2' and '4' in the function call 'add(2, 4)'. One arrow points up to the parameter 'a' in the function definition 'def add(a, b):', and the other points up to the parameter 'b'.

In function calling, the actual arguments 2 and 4 are sent to the formal parameters a and b respectively.

Note that the returned value 6 is sent back to the call expression add(2, 4).

This value add(2, 4) is stored in the variable a.

The variable a is then printed to the console.



# Returned Value

A returned value from a function should be stored, printed or used in another calculation. Be careful to avoid the error explained below!

```
def add(a, b):  
    return a + b
```

```
a = add(2, 4)          # returned value 6 is stored in a.  
print(a)               # 6  
b = add(1, 2) - 3      # returned value 6 used in a calculation.
```

```
add(4, 6)              # returned value 10 is neither stored nor printed  
                        # this value is lost! This is a common error!  
                        # This line of code effectively does nothing.
```

# Keyword Arguments

Input arguments can be specified by name (keyword arguments). In this case, the order does not matter. Using keyword arguments is encouraged since it makes code clear and flexible.

```
def add(a, b):  
    return a + b
```

The following are all equivalent:

```
x = add(1, 3)
```

```
y = add(a=1, b=3)
```

```
z = add(b=3, a=1)
```

# Functions

If keyword arguments and positional arguments are used together, keyword arguments must follow positional arguments.

```
def add(a, b, c):  
    return a + b + c
```

The following are all equivalent:

```
x = add(1, 3, 5)  
y = add(1, b=3, c=5)  
z = add(1, c=5, b=3)
```

The following gives an error:

```
w = add(a=1, 3, c=5)
```

# Functions

Functions can take optional keyword arguments. These are given default values. Their default values are used if a user does not specify these inputs when calling the function. Default values must come after positional arguments in the function signature.

```
def subtract(a, b=0):  
    return a - b
```

a = 2, b defaults to 0

```
print(subtract(2))          # 2
```



# Functions

Functions can take optional keyword arguments. These are given default values. Their default values are used if a user does not specify these inputs when calling the function. Default values must come after positional arguments in the function signature.

```
def subtract(a, b=0):  
    return a - b
```

*a = 1, b is overwritten; b = 3*

```
print(subtract(2))           # 2  
print(subtract(1, 3))       # -2
```

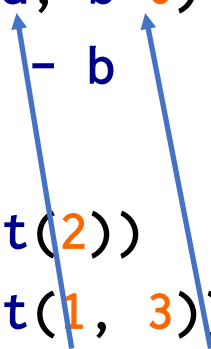
# Functions

Functions can take optional keyword arguments. These are given default values. Their default values are used if a user does not specify these inputs when calling the function. Default values must come after positional arguments in the function signature.

```
def subtract(a, b=0):  
    return a - b
```

*a = 2, b is overwritten; b = 5*

```
print(subtract(2))           # 2  
print(subtract(1, 3))       # -2  
print(subtract(2, b=5))     # -3
```



# Functions

Functions can take optional keyword arguments. These are given default values. Their default values are used if a user does not specify these inputs when calling the function. Default values must come after positional arguments in the function signature.

```
def subtract(a, b=0):  
    return a - b
```

*a = 6, b is overwritten; b = -1*

```
print(subtract(2))           # 2  
print(subtract(1, 3))       # -2  
print(subtract(2, b=5))     # -3  
print(subtract(b=-1, a=6))  # 7
```

# Python Script

A Python script is executed line by line top to bottom.

Function definitions are packaged into an executable unit to be executed later. The code within a function definition executes only when invoked by a caller.

In addition, variables and parameters **defined in a function is local to that function and is hidden from code outside of the function definition.**



# Flow of a Program

```
x = 2
def fun():
    x = 10
    print(x)
print(x)
fun()
print("Good bye!")
```

Diagram illustrating the flow of a program:

- A red number **2** is placed above the `def fun():` line.
- A red number **3** is placed to the right of the `print(x)` line inside the function.
- A blue arrow points from the `def fun():` line down to the `print(x)` line inside the function.
- A blue arrow points from the `print(x)` line inside the function up to the `print(x)` line outside the function.
- A blue arrow points from the `fun()` line down to the `print("Good bye!")` line.

A procedure or function call interrupts the sequential execution of statements, causing the program to execute the statements within the procedure before continuing.

Once the last statement in the procedure (or a return statement) has executed, flow of control is returned to the point where the procedure was called.

Output:

2

10

Good bye!

Note in the code, 2 is printed before 10.

The x variable defined outside of the function is different from the x variable defined inside of the function.

# main.py

```
x = 2
print("1. x =", x)
def fun1():
    x = 10
    print("2. x =", x)
print("3. x =", x)
def fun2():
    x = 20
    print("4. x =", x)
print("5. x =", x)
fun1()
fun2()
print("6. x =", x)
```

What's the output? Remember that code is executed top to bottom. However, function code only executes when the function is called.

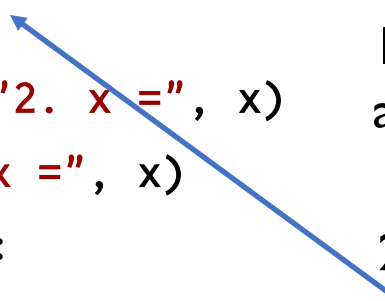
Variables defined inside a function are different than those defined outside of a function even if they have the same name.

Output:

```
1. x = 2
3. x = 2
5. x = 2
2. x = 10
4. x = 20
6. x = 2
```

# main.py

```
x = 2
print("1. x =", x)
def fun1():
    x = 10
    print("2. x =", x)
print("3. x =", x)
def fun2():
    x = 20
    print("4. x =", x)
print("5. x =", x)
fun1()
fun2()
print("6. x =", x)
```

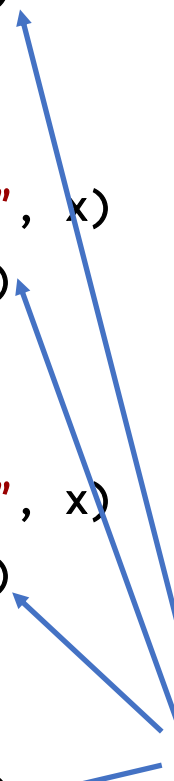


This example illustrates how functions protect its local variables. Things to note:

- 1) Function definitions are not executed until they are explicitly called.
- 2) Two different functions can use local variables named `x`, and these are two different variables that have no influence on each other. This includes parameters.

# main.py

```
x = 2
print("1. x =", x)
def fun1():
    x = 10
    print("2. x =", x)
print("3. x =", x)
def fun2():
    x = 20
    print("4. x =", x)
print("5. x =", x)
fun1()
fun2()
print("6. x =", x)
```



This example illustrates how functions protect its local variables. Things to note:

- 1) Function definitions are not executed until they are explicitly called.
- 2) Two different functions can use local variables named `x`, and these are two different variables that have no influence on each other. This includes parameters.
- 3) The `x` variable defined outside of `fun1()` and `fun2()` is not affected by the code inside of those functions. (`x = 2`)

# Scope

The **scope** of a variable refers to the context in which that variable is visible/accessible to the Python interpreter.

A variable has **file scope** if it is visible to all parts of the code contained in the same file.

A variable defined inside a function or as input arguments has **restricted scope** – they can only be accessed within the function.

Python is more liberal compared to Java and C++ in terms of scoping rules. In most cases, variables have file scope.

# Scope

```
a = 2 # a has file scope
```

```
def fun(b):
```

```
    c = b + 1 # b and c both have restricted scope
```

```
    return c
```

```
g = fun(3)
```

```
print(g) # 4
```

```
print(c) # error, this is outside of scope of c, c not defined
```

```
if a % 2 == 0:
```

```
    f = 5 # f has file scope
```

```
print(f) # 5
```

# Python Program Template

```
# declare and initialize global variables with file scope, these  
# variables exist everywhere in the rest of the file including inside  
# functions.
```

```
x = 3
```

```
# function definitions
```

```
def func1(...):
```

```
    ...
```

```
def func2(...):
```

```
    ...
```

```
# program logic flow starts here
```

```
# ask for user inputs, call functions above, etc..
```

```
a = func1()
```

```
print(a)
```

From now on, when we write a  
program, we will use this template.

# Writing a Simple Program: Quadratic Roots

Let's write a full program that asks the user for three integers  $a$ ,  $b$  and  $c$  which represent the coefficients of a quadratic function of the form

$f(x) = ax^2 + bx + c$  and outputs the number of real zeroes or roots of  $f(x)$ .

```
def num_of_roots(a, b, c):
    discriminant = b ** 2 - 4 * a * c
    if discriminant > 0:
        return 2
    elif discriminant < 0:
        return 0
    else:
        return 1

a = float(input('Enter a:'))
b = float(input('Enter b:'))
c = float(input('Enter c:'))
numroots = num_of_roots(a, b, c)
print("This quadratic has", numroots, "real root(s).")
```



# Lab 1: Day Of the Week

Create a new repl on repl.it. Write a program that outputs the day of the week for a given date! Your program must use the program template discussed in this lecture. It should include the main function and the `day_off_week` function below.

Given the month,  $m$ , day,  $d$  and year  $y$ , the day of the week (Sunday = 0, Monday = 1, ..., Saturday = 6)  $D$  is given by

$$y_0 = y - (14 - m)/12$$

$$x_0 = y_0 + y_0/4 - y_0/100 + y_0/400$$

$$m_0 = m + 12 \times ((14 - m)/12) - 2$$

$$D = (d + x_0 + 31 \times m_0/12) \bmod 7$$

**Note: the `/` operator from the above equations is floor division `//` in Python. The mod operator is `%`.**

**Use the template on the next page.**

# Lab I: Day Of the Week

Use the following template.

```
def compute_day(month, day, year):  
    """ This function computes the values given from the previous slide  
        and returns an integer in the set {0,1,...,5,6}.  
    """  
  
def day_of_week(d):  
    """ Given d which computed from compute_day above. This function returns  
        a string according to the value of d: "Sunday" for 0, "Monday" for 1,  
        etc..  
    """  
  
# ask users for month, day and year  
# call compute_day and day_of_week above  
# print out day of the week.
```

# Lab 1: Day Of the Week

Your program should have output similar to the following:

Enter month: 10

Enter day: 27

Enter year: 2020

Day of the week: Tuesday

And try entering your birthday and test your parents!

# References

- 1) Vanderplas, Jake, A Whirlwind Tour of Python, O'reilly Media.
- 2) Halterman, Richard, Fundamentals of Python Programming.