Introduction to Python

Functions

Topics

1) Functions
   a) Positional Arguments
   b) Keyword Arguments
   c) Default Values
2) Installing and Running a Python Script
3) Scope of a variable
4) Execution of a Script
5) main()

Functions

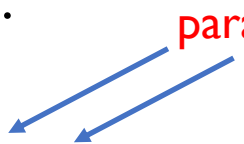
One way to organize Python code and to make it more readable and reusable is to factor out useful pieces into reusable *functions*.

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

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

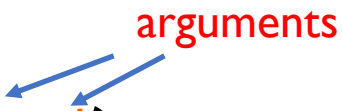
Functions

The arguments of a function are the inputs of the function. By default, arguments are positional. A function can "return" an answer, which is the output of the function.




```
In [1]: def add(a, b):
            return a + b
```

Function definition.




```
In [2]: add(2, 4)
Out [2]: 6
```

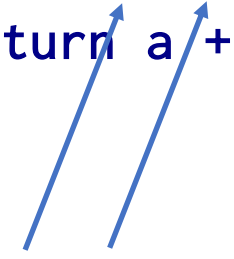
Function calling.

```
In [3]: add(2) # too few arguments
In [4]: add(2, 4, 6) # too many arguments
```

Functions Arguments (input)

```
In [1]: def add(a, b):
            return a + b

In [2]: add(2, 4)
Out [2]: 6
```



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

Function Return (output)

```
In [1]: def add(a, b):
            return a + b

In [2]: answer = add(2, 4)
In [3]: print(answer)
6
```

When a function returns a value, the returned value is the value of the entire function call add(2,4).

Functions

Let's write a simple function absolute_value which returns the absolute value of a given number.

```
In [1]: def absolute_value(n):
            if n >= 0:
              return n
            else:
              return -n

In [2]: absolute_value(23)

Out [2]: 23

In [3]: absolute_value(-10)

Out [3]: 10

In [4]: abs(-3)

Out [4]: 3
```

You write the function definition once.
But you can call it many times!
This is code reuse.

Python already has a built-in function for called abs() which returns the absolute value of a number.

Functions

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.

```
In [1]: def add(a, b):
            return a + b
```

The following are all equivalent:

```
In [2]: add(1, 3)
In [3]: add(a=1, b=3)
In [4]: add(b=3, a=1)
```

Functions

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

```
In [1]: def add(a, b, c):
            return a + b + c
```

The following are all equivalent:

```
In [2]: add(1, 3, 5)
In [3]: add(1, b=3, c=5)
In [4]: add(1, c=5, b=3)
```

The following gives an error:

```
In [5]: 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.

```
In [1]: def subtract(a, b=0):
            return a - b
In [2]: subtract(2)
Out [2]: 2

In [3]: subtract(1, 3)
Out [3]: -2

In [4]: subtract(2, b=5)
Out [4]: -3

In [5]: subtract(b=-1, a=6)
Out [5]: 7
```

We have seen the use of optional keyword arguments before.

```
print(*objects, sep=' ', end='\n')
```

any number of **positional arguments**

optional keyword argument specifies separator character; default value is a space

specify end character; defaults to newline

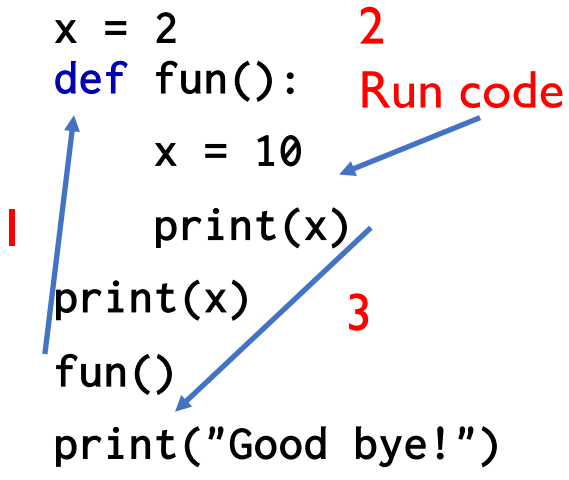
```
print("hello", "Mike")  # use default values
print("goodbye", "Sarah", end=".", sep=":")
```

note that the order can be switched.

hello Mike

goodbye:Sarah.

Flow of a Program



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 immediately following where the procedure was called.

Writing A Complete Program

Thus far, we have used the IPython console to write one line of code or block of code at a time.

As the complexity of our program increases, we like to decompose our programs into small reusable components(functions, objects) and combine them in a logical way to form a complete program.

When you create a new repl on repl.it, notice that your code lives inside of the "main.py" file. A file that ends in .py is a Python **script or module**: a text file that contains Python code.

Writing A Complete Program

Offline, you can create Python scripts with any simple text editor. For example, on Windows, you can use notepad. On a Mac, textEdit.

However, it useful to use a more sophisticated "integrated development environment"(IDE) to have features like autocomplete, ability to access docs strings.

For us, we have used repl.it IDE. But there are great offline ones including PyCharm and very popular Visual Studio Code.

Note: Do not use Word to write code. Word will silently change certain characters like " and will cause errors in your code.

Installing Python

To run code locally on computer, you need a Python interpreter.

It is highly recommended that you download and install the Anaconda distribution which includes the official CPython interpreter, useful packages for scientific computing like NumPy and SciPy, the conda package manager, the Jupyter Notebook as well some other IDEs(VSCode).

Running Code Locally on Computer

Once you have the Python interpreter installed, you can run code locally on you computer.

Navigate to the director(folder) where your script(e.g. "main.py") lives.

On the terminal(Mac) or command prompt(Win), type:

```
python main.py
```

The output will appear on your terminal.

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.

main.py

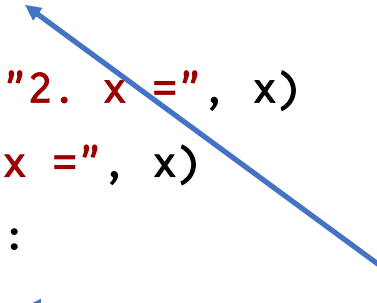
```
x = 2
print("1. x =", x)
def fun1():
    x = 10
    print("2. x =", x)
print("3. x =", x)
def fun2():
    x = 20
    print("4. x =", x)
print("5. x =", x)
fun1()
fun2()
print("6. x =", x)
```

What's the output?

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)
```



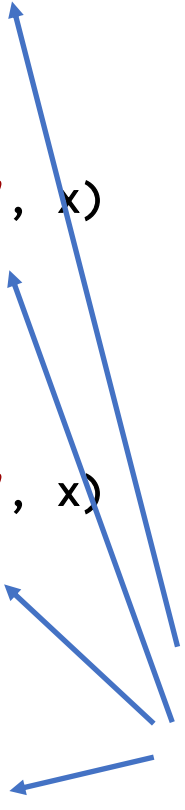
This example illustrates how functions protect its local variables. Things to note:

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
```

Writing a Simple Program

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(numroots)
```

It is common for programmers to write a main controlling function that calls other functions to accomplish the task of the program.

```
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
def get_int():
        return int(input("Enter a number: "))
def main():
        a = get_int()
        b = get_int()
        c = get_int()
        print(num_of_roots(a, b, c))
main()
```

In other programming languages like Java and C++, the main program is REQUIRED to be called main().

the program starts running here.

```
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
def get_int():
        return int(input("Enter a number: "))
def main():
        a = get_int()
        b = get_int()
        c = get_int()
        print(num_of_roots(a, b, c))

if __name__ == 'main':
        main()
```

You might also see this version of calling main().
It has to do with importing vs. running a script.
We won't worry too much about this for now.

Python Program Template

```
# declare and initialize global variables with file scope
…
# function definitions
def func1(…):
        …
def func2(…):
        …

# main() function calls above functions to accomplish task of application
def main():
        …
main()
# OR
# if __name__ == 'main':
#      main()
```

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

Lab 1: Day Of the Week

Create a new repl on replit. Write a program that outputs the day of the week for a given date! You program must 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:

$$
\begin{aligned}
y_0 &= y - (14 - m)/12 \\
x_0 &= y_0 + y_0/4 - y_0/100 + y_0/400 \\
m_0 &= m + 12 \times ((14 - m)/12) - 2 \\
\mathcal{D} &= (d + x_0 + 31 \times m_0/12) \bmod 7
\end{aligned}
$$

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

Your program needs the following method and the main function.

```
def day_of_week(m, d, y):
    # your code implementation
```

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.