Introduction to Python

Functions

Topics

1) Functions
   a) Positional Arguments
   b) Keyword Arguments
   c) Default Values

2) Built-in Functions

3) Installing and Running a Python Script

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

```python
def function_name(arguments):
    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):        Function
                              definition.
            return a + b

In [2]: add(2, 4)
                              Function
Out [2]: 6
                              calling.
In [3]: add(2) # too few arguments

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

Functions Parameters (input)

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


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

In function calling, the actual parameters 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]: add(2, 4)
Out [2]: 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 is_prime which returns whether a number is prime and see how putting code into a function is very useful.

```
In [1]:    def is_prime(n):
                count = 0
                for x in range(1, n+1):
                    if n % x == 0:
                        count += 1
                return count == 2

In [2]: is_prime(17321)
Out [2]: True

In [3]: is_prime(15879)
Out [3]: False
```
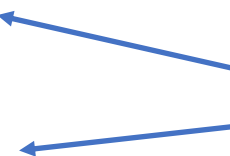
You write the function definition once.
But you can call it many times!
This is code reuse.

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)
```

```
In [5]: add(5, 10)
Out [5]: 15

In [6]: add([1,2], [5,6])
Out [5]: [1,2,5,6]
```

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.

```python
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
```

Built-in Functions

There are many useful built-in functions. These functions can be called directly without importing any modules.

abs()	returns the absolute value of a numeric value.
min(), max()	returns the minimum and maximum of an iterable, respectively.
sorted(object, reverse=False)	returns a sorted iterable. b=By default, sort the iterable in ascending order unless reverse=True.
sum()	returns the sum of an iterable.

Built-in Functions

```
In [1]: abs(-3)
Out [1]: 3
In [2]: lst = [4, -3, 5, 2]
In [3]: min(lst)
Out [3]: -3
In [4]: max(lst)
Out [4]: 5
In [5]: sum(lst)
Out [5]: 8
```

Built-in Functions

```
In [1]: lst = ["Mike", "Sarah", "Andy"]

In [2]: sorted(lst)
Out [2]: ["Andy", "Mike", "Sarah"]

In [3]: sorted(lst, True)
Out [3]: ["Sarah", "Mike", "Andy"]

In [4]: sorted(lst, reverse=True)
Out [3]: ["Sarah", "Mike", "Andy"]
```
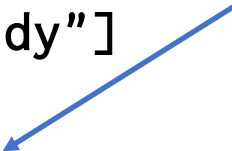
One of Python's philosophy is "explicit is better than implicit"; it's better to use the keyword argument reverse=True here.

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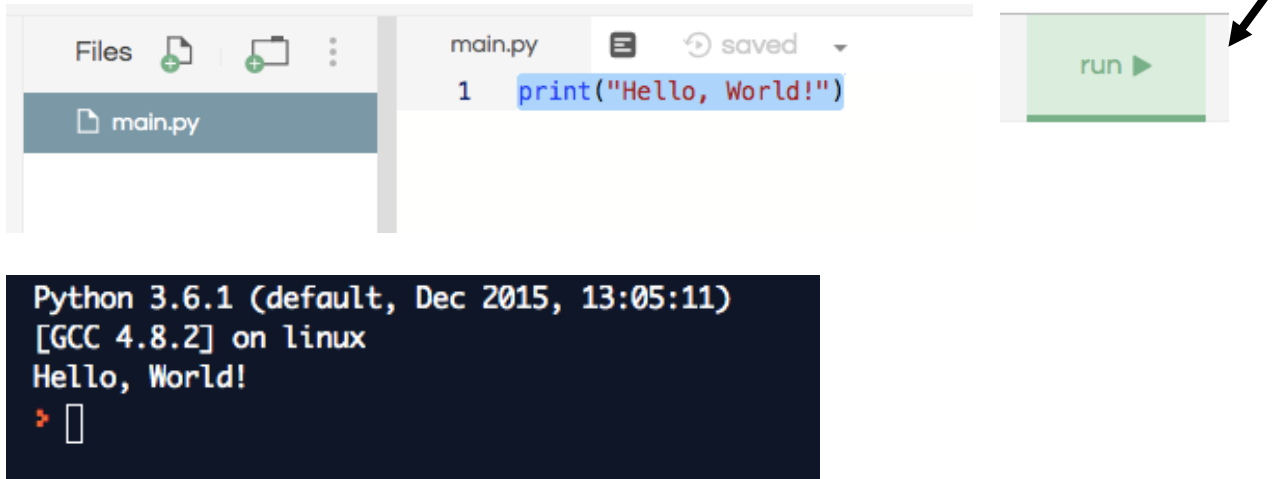
Note: Do not use Word to write code. Word will silently change certain characters like " and will cause errors in your code.

Running A Script on repl.it

```python
print("Hello, World!")
```

repl.it: Create a new repl and pick Python as the language, name the repl.

Type in the code in the file main.py. Click on run.

Files

main.py

main.py saved

1 print("Hello, World!")

run ▶

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
Hello, World!
>
```

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

```python
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)
```
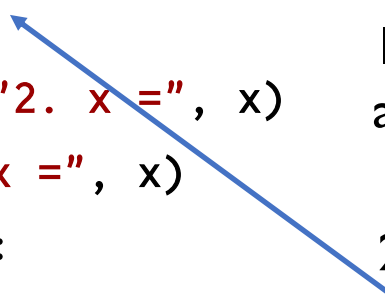
What's the output?

Output:
1. x = 2
3. x = 2
5. x = 2
2. x = 10
4. x = 20
6. x = 2

main.py

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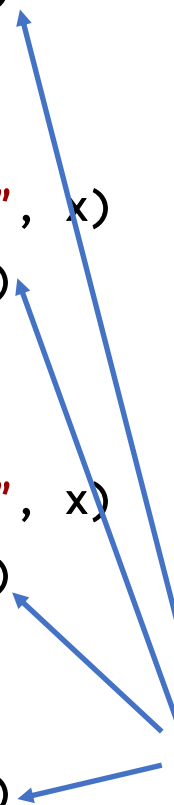
This example illustrates how functions protect its local variables. Things to note:

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

```python
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 = 1 # a has file scope
def fun(b):
    c = b + 1  # b and c both have restricted scope
    return c


lst = [2 * i for i in range(10)] # i has restricted scope


if a % 2 == 0:
    f = 5  # f has file scope


for j in range(5):
    g = 5 * j # g and i have file scope
```

Immutable Objects in Parameters

```python
def fun(a):
    a = 0


x = 1
fun(x)
print(x) # 1
```

Initially, a and x reference the same object: 1. However since int objects are immutable, this function creates a new object and a references the new object 0. The reference of x is unchanged.

Mutable Objects in Parameters

```python
def fun2(a):
    a.append(0)


x = [1,2,3]
fun2(x)
print(x) # [1,2,3,0]
```

Both a and x reference the same list object. Since list objects are mutable, modifying a will also modify x since both reference the same list.

Function Example

Write a function that accepts two integer inputs and returns their greatest common divisor(gcd). For example gcd(24, 16) = 8.

```python
def gcd(a, b):
    min = b if a > b else a
    factor = 1
    for i in range(1, min + 1):
        if a % i == 0 and b % i == 0:
            factor = i
    return factor
```

Writing a Simple GCD Program

Let's write a full program that asks the user for two integers and print out the gcd.

main.py

```python
def gcd(a, b):
    min = b if a > b else a
    factor = 1
    for i in range(1, min + 1):
        if a % i == 0 and b % i == 0:
            factor = i
    return factor
num1 = int(input('Please enter an integer: '))
num2 = int(input('Please enter another integer: '))
print(gcd(num1,num2))
```

Writing a Simple GCD Program

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

main.py

```python
def gcd(a, b):
        min = b if a > b else a
        factor = 1
        for i in range(1, min + 1):
                if a % i == 0 and b % i == 0:
                        factor = i
        return factor
def get_int():
        return int(input("Enter a number: "))
def main():
        num1 = get_int()
        num2 = get_int()
        print(gcd(num1, num2))
main()
```

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

Writing a Simple GCD Program

main.py

```python
def gcd(a, b):
        min = b if a > b else a
        factor = 1
        for i in range(1, min + 1):
                if a % i == 0 and b % i == 0:
                        factor = i
        return factor
def get_int():
        return int(input("Enter a number: "))
def main():
        num1 = get_int()
        num2 = get_int()
        print(gcd(num1, num2))


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 func1(…):

        …


# 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.

References

1) Vanderplas, Jake, A Whirlwind Tour of Python, O'reilly Media.