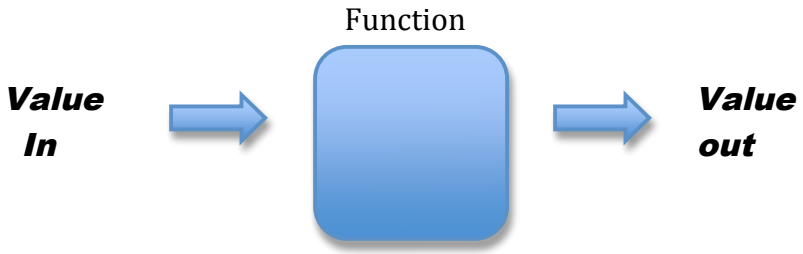
Programming with Functions

In an earlier section, we discussed the idea that a program consists of an ordered sequence of commands (in a specific language). When we want to write a program to solve a large complicated problem, one of the keys is to break the problem up into smaller, manageable, and reusable pieces. We do this through the use of **functions**.

Mathematicians frequently work with functions such as $f(x) = x$, $g(x) = x^2$, $h(x) = \sin x$, and others. The idea of a mathematical function is that the value of one variable (or input) completely determines another value (the value, or output). A function is like a machine – you input some value, the function performs some calculation or operation, and then gives some value back out.



When you write a program, you will also work with functions, except they will look slightly different than these mathematical functions.

Let's talk about the different parts of the functions above. To begin with, we typically refer to functions by their **names**, which, in the functions above, are *f*, *g*, and *h*, respectively. In math, we often use single letters to represent names of functions, but there is no reason we couldn't use words to create more meaningful names, such as $Identity(x) = x$, or $Square(x) = x^2$ instead of *f* or *g*. In programming, it is very useful to use meaningful names.

Now, what about the part of the function in the parentheses? In math, the *x* is usually referred to as the independent variable. The actual value of this variable is what determines the value that gets output from the function. In programming, we will refer to *x* as the **parameter** of the function. When we call (or *invoke*) the function, we will need to specify a value for *x*, such as $Square(2)$. This value 2 is called the **argument** of the function.

Helpful Hint

A *parameter* is the part of the definition of the function that indicates what the function must receive when it is used, or invoked, and an *argument* is the value that is specified when you are using, or *invoking*, a function.

Let's see what the function $Square(x)$ would look like in Python:

```
def Square (x):          (1)
    return x * x         (2)
```

Line (1) is referred to as the **function definition**. Line (2) is the **return statement**, and in this example, also the entire **body of the function**. The body of the function consists of all of the statements that should be executed when the function gets called. In Python, the body of the function gets indented. If the function needs to return a value, the return statement must be the *last* statement in the body. Nothing gets executed after the return statement.

In general, when we want to define a function in Python, we must start with the keyword `def`, followed by the name we want to give the function, followed by parentheses and lastly, a colon. Then we list any parameters that the function needs inside the parentheses, separated by commas.

Example: A function to calculate the average of two numbers

```
def average (num1, num2):
    avg = (num1 + num2)/2.0
    return avg
```

We would write this function in the program area (the top/white part) of JES. We could then invoke (call) this function in the command area (the bottom/black) of JES, by typing a statement such as `average(3, 4)`. It would look like this in JES:

```
>>> average (3, 4)                    (*)
3.5
```

We could also use variables to hold the values we would like to take the average of, such as:

```
>>> a = 5
>>> b = 7
>>> averageOfNums = average(a, b)      (**)
```

Notice that in this example, the actual value of the average doesn't appear on the screen. It is stored in the variable `averageOfNums`. If we wanted to see that value, we could add an additional statement:

```
>>> a = 5
>>> b = 7
>>> averageOfNums = average(a, b)
>>> print averageOfNums
6
```

Exercise: Identify the following parts of the `average` function in the previous example:

a) Name of the function
b) Parameters
c) Return statement
d) Argument values in the call at (*)
e) Argument values in the call at (**)

In the `average` function example, it did not matter which order the parameters or arguments were specified. The two calls `average(3, 4)` and `average(4, 3)` would both return the same value. (Test it to convince yourself!) This is not the case with most functions. Consider the following example that demonstrates this:

```
def mathExpression(num1, num2):
    value = num1 * 2 + num2
    return value
```

What value is returned from this function with the call
`mathExpression(5, 7)`?

What value is returned from this function with the call
`mathExpression(7, 5)`?

Are these values the same? (You should have found that they do NOT return the same values!)

So, in general, you can specify parameters in a function in any order, but when you call the function, the values of the arguments must match the order in which the parameters were specified.

Helpful Hint

The order of the arguments in a call to a function must match the order of the parameters in the function definition.

Here is an additional example to demonstrate this idea:

```
def doStuff(message, num):
    print message
    value = num*2
    return value
```

What happens when we call this function with `doStuff("Hello", 10)`? We see the word `Hello` printed to the screen and the value 20 gets returned. Now what happens when we call this function with `doStuff(10, "Hello")`? Will *anything* happen? If we walk through this, we find that the `message` parameter gets the value 10 and the `num` parameter gets the value `Hello` when the function is called. The print statement will print the value 10 to the screen, but then the program will halt execution (*i.e.* **crash**) when it tries to multiply a non-numerical value by a numerical value. This function won't work if the order of the arguments does not match the order of the parameters.

In the first mini-lab, we learned how to invoke functions such as `pickAFile`, `makePicture`, and `makeSound`. We saw that `pickAFile` did not require any parameters, although `makePicture` and `makeSound` each required a filename as a parameter.

We can create new functions by combining functions we already know about. For example, let's write a function that would allow the user to pick an image file, turn it into a picture object, and then display the picture. (These are all things we did in the first mini-lab.) It would look like the following:

```
def pickAndShow():
   myFile = pickAFile()
   myPict = makePicture(myFile)
   show(myPict)
```

It is now time to explore some functions in Python. In this mini-lab, you will gain practice with variables, defining, and using functions in Python, and the mechanism for saving and loading files in JES.

Mini-Lab: Exploring Functions