Introduction to Programming in Python

Functions

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: June 4, 2021 at 11:04

Function

We've seen lots of system-defined functions; now it's time to define our own.

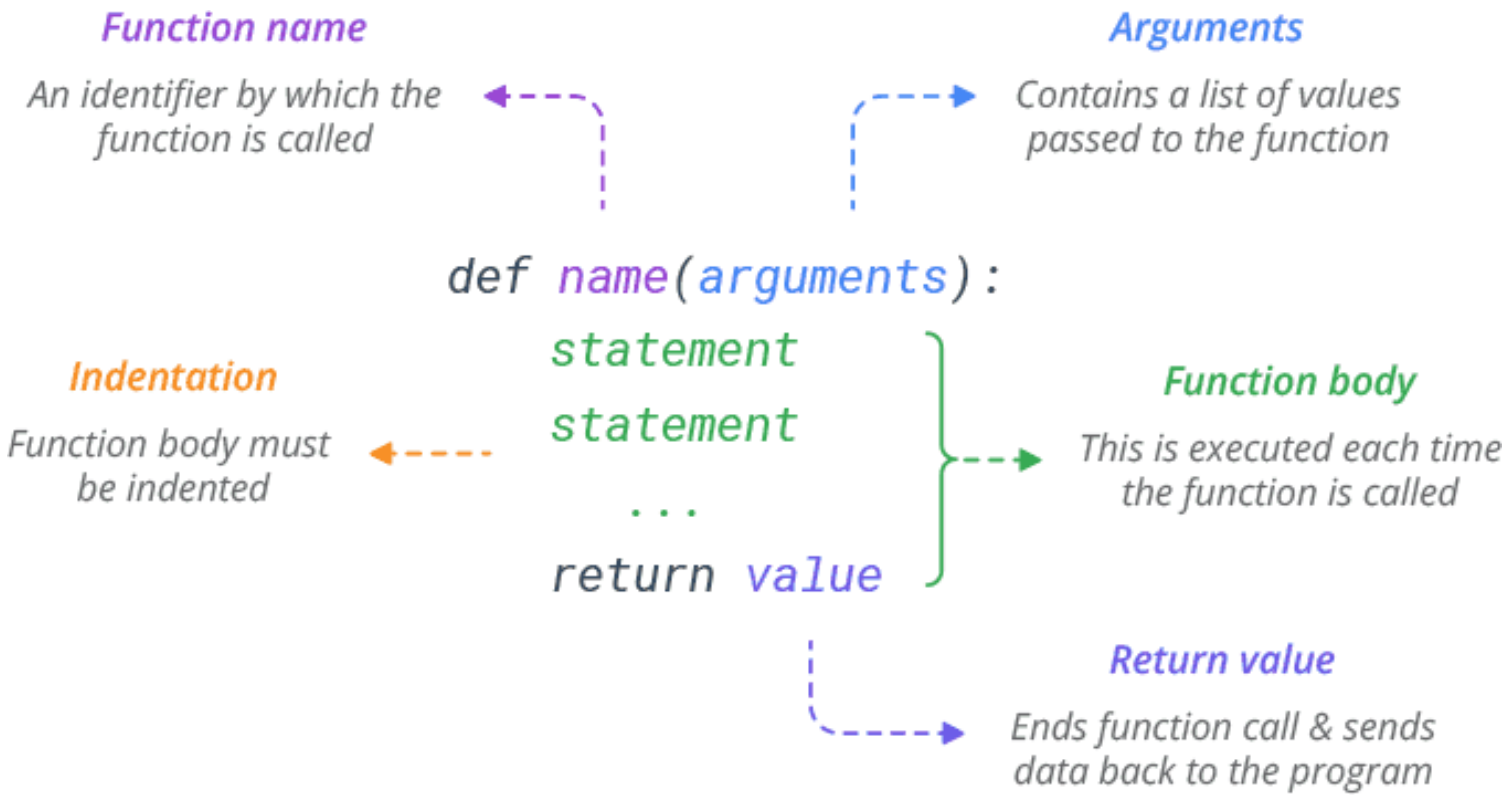
General form:

```
def functionName( list of parameters ):   # header
    statement(s)                          # body
```

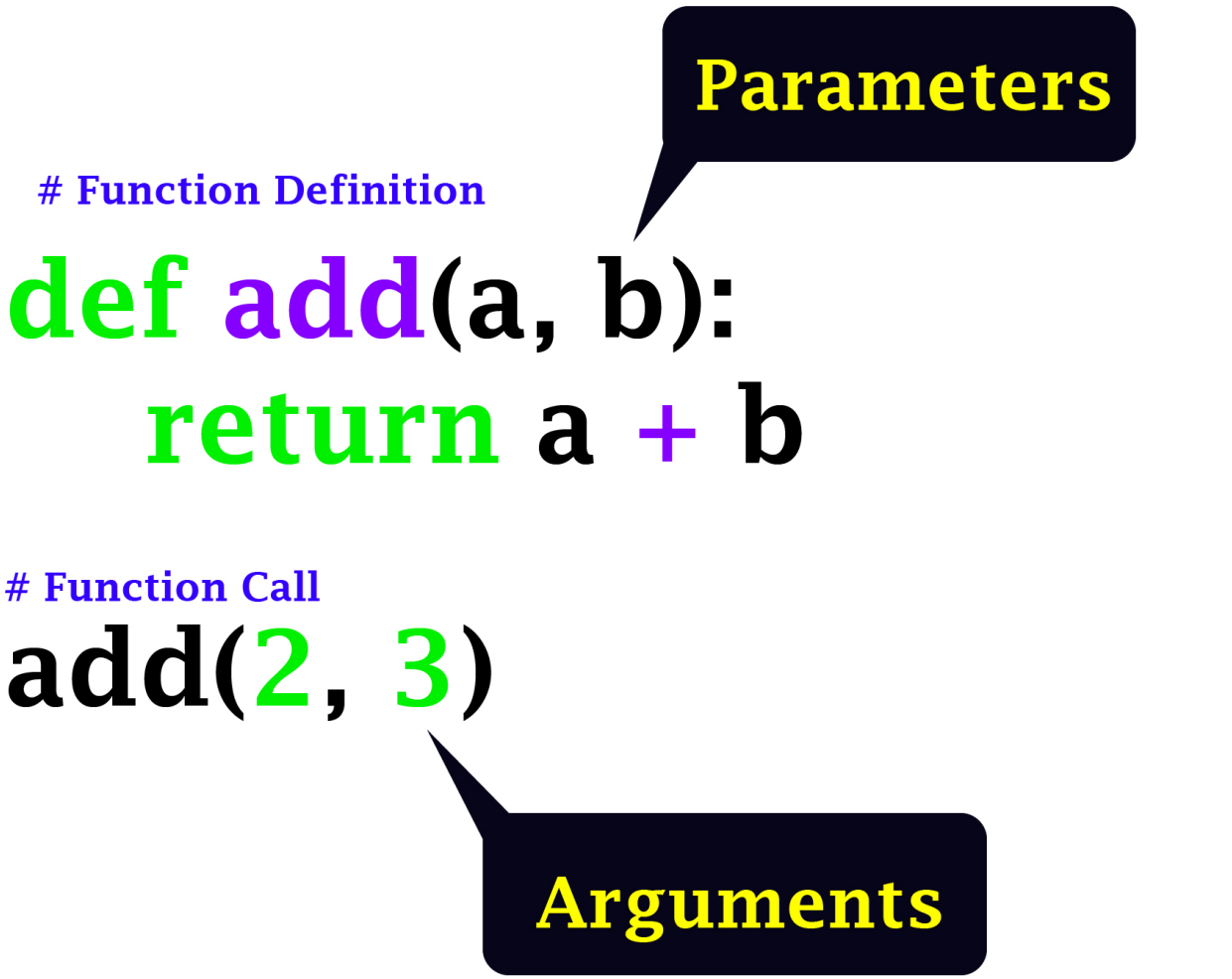
Meaning: a function definition defines a block of code that performs a specific task. It can reference any of the variables in the list of parameters. It may or may not return a value.

The parameters are **formal parameters**; they stand for arguments passed to the function later.

Functions


Function name
An identifier by which the function is called
Arguments
Contains a list of values passed to the function
def name(arguments):
statement
statement
...
return value
Indentation
Function body must be indented
Function body
This is executed each time the function is called
Return value
Ends function call & sends data back to the program

Calling a Function



getKT.com

Function Example

Suppose you want to add up the integers 1 to n.

In file `functionExamples.py`:

```
def sumToN( n ):
    """ Sum the integers from 1 to n. """
    sum = 0                              # identity element for +
    for i in range(1, n + 1 ):           # Why n+1?
        sum += i
    # hand the answer to the calling environment
    return sum
```

Notice this defines a *function* to perform the task, but *doesn't actually perform the task*. We still have to call/invoke the function with specific arguments.

```
>>> from functionExamples import *
>>> sumToN( 10 )
55
>>> sumToN( 1000 )
500500
>>> sumToN(1000000)
500000500000
```

Some Observations

```
def sumToN( n ):                    # function header
    ....                            # function body
```

Here `n` is a *formal parameter*. It is used in the definition as a place holder for an *actual parameter* (e.g., 10 or 1000) in any specific call.

`sumToN(n)` *returns* an `int` value, meaning that a call to `sumToN` can be used anyplace an `int` expression can be used.

```
>>> print( sumToN( 50 ) )
1275
>>> ans = sumToN( 20 ) + sumToN( 30 )
>>> print( ans )
675
```

Functional Abstraction

Once we've defined `sumToN`, we can use it almost as if were a primitive in the language without worry about the details of the definition.

We need to know what it does, but don't care anymore how it does it!

This is called **information hiding** or **functional abstraction**.

Another Way to Add Integers 1 to N

Suppose later we discover that we could have coded `sumToN` more efficiently:

```
def sumToN( n ):
    """ Sum the integers from 1 to n. """
    return ( n * (n+1) ) // 2
```

Because we defined `sumToN` *as a function, we can just swap in this definition without changing any other code.*

```
>>> sumToN(10)
55
>>> sumToN( 1000000000000 )
500000000000500000000000
```

Return Statements

When you execute a `return` statement, you return to the calling environment. You may or may not return a value.

General forms:

```
return
return expression
```

A `return` that doesn't return a value actually returns the constant `None`.

Every function has an *implicit* return at the end.

```
def printTest ( x ):
  print( x )
```

About return

In file `returnExamples.py`:

```
def printSquares():
    """ Compute and print squares until 0 is entered
        by the user. """
    while True:
        num = int(input("Enter an integer or 0 to exit: "))
        if ( num != 0 ):            # "if num:" works
            print( "The square of", num, "is:", num ** 2 )
        else:
            return                  # no value is returned

printSquares()
```

This doesn't return a value, but accomplishes it's function by the "side effect" of printing.

About return

```
> python returnExamples.py
Enter an integer or 0 to exit: 7
The square of 7 is: 49
Enter an integer or 0 to exit: -12
The square of -12 is: 144
Enter an integer or 0 to exit: 0
>
```

A function that "doesn't return a value" actually returns the constant `None`.

Some More Function Examples

Suppose we want to multiply the integers from 1 to n:

```
def multToN( n ):
    """ Compute the product of the numbers from 1 to n. """
    prod = 1                    # identity element for *
    for i in range(1, n+1):
        prod *= i
    return prod
```

Convert fahrenheit to celsius:

```
def fahrToCelsius( f ):
""" Convert fahrenheit temperature value to celsius
    using formula: C = 5/9(F-32). """
    return 5 / 9 * ( f - 32 )
```

Or celsius to fahrenheit:

```
def celsiusToFahr( c ):
    """ Convert celsius temperature value to fahrenheit
        using formula: F = 9/5 * C + 32. """
    return 9 / 5 * c + 32
```

Some Function Exercises

Write functions to return:

1. `areaOfSquare( side )`: The area of a square with the given side.
2. `perimeterOfSquare( side )`: The permiter of a square with the given side.
3. `areaOfCircle( rad )`: The area of a circle with the given radius (πr^2).
4. `circumferenceOfCircle( rad )`: The circumference of a circle with the given radius ($2\pi r$).

Remember: you can get π by importing math and using `math.pi`.

Function Exercises

In file `figures.py`:

```
import math

def areaOfSquare( side ):
    return side ** 2

def perimeterOfSquare( side ):
    return 4 * side

def areaOfCircle( rad ):
    return math.pi * ( rad ** 2 )

def circumferenceOfCircle( rad ):
    return 2 * math.pi * rad
```

If you put your code into a file called `filename.py` (where `filename` is whatever you like), you're defining a *module* called *filename* that you can import for use anywhere.

Function Exercises

Running our functions:

```
> python
Python 3.6.9 (default, Jan 26 2021, 15:33:00)
>>> from figures import *
>>> areaOfSquare( 3.4 )
11.559999999999999
>>> areaOfSquare( 2 )
4
>>> perimeterOfSquare( 5.2 )
20.8
>>> areaOfCircle( 5.0 )
78.53981633974483
>>> circumferenceOfCircle( 5.0 )
31.41592653589793
>>>
```

Exercise: Now define functions for the area and perimeter of a rectangle, the volume of a sphere ($\frac{4}{3}\pi r^3$).

Functions and Return Values

Functions can return a value or not. A function that doesn't return a value is sometimes called a **procedure**.

Actually, every function returns a value, but some return the special value `None`.

A function that doesn't return a value may still do useful work, for example, by printing a table of values.

Positional Arguments

This function has four formal parameters:

```
def functionName ( x1, x2, x3, x4 ):
    < body >
```

Any call to this function should have exactly four actual arguments, which are matched to the corresponding formal parameters:

```
    functionName( 9, 12, -3, 10 )
    functionName( 'a', 'b', 'c', 'd' )
    functionName( 2, "xyz", 2.5, [3, 4, 5] )
```

This is called using **positional** arguments.

Default Parameters

You can also specify **default arguments** for a function. If you don't specify a corresponding actual argument, the default is used.

```
def printRectangleArea( width = 1, height = 2 ):
    area = width * height
    print("width: ", width, "\theight: ", height, \
          "\tarea:", area)

printRectangleArea()                          # use defaults
printRectangleArea(4, 2.5)                    # positional args
printRectangleArea(width = 1.2)               # default height
printRectangleArea(height = 6.2)              # default width
```

Using Defaults

```
> python RectangleArea.py
width:  1       height:  2       area: 2
width:  4       height:  2.5     area: 10.0
width:  1.2     height:  2       area: 2.4
width:  1       height:  6.2     area: 6.2
```

Notice that you can mix default and non-default arguments, but must define the non-default arguments first.

```
def email (address, message = ""):
```

Returning Multiple Values

The Python `return` statement can also return multiple values. In fact it returns a *tuple* of values.

```
def multipleValues ( x, y ):
  return x + 1, y + 1

print( "Values returned are: ",  multipleValues ( 4, 5.2 ))

x1, x2 = multipleValues( 4, 5.2 )
print( "x1: ", x1, "\tx2: ", x2 )
```

```
Values returned are:  (5, 6.2)
x1:  5  x2:  6.2
```

Generating Random Passwords

Remember our example for generating random passwords. Let's write a function that will generate a password of length k.
In file `Password2.py`:

```
import random

def generatePassword( passwordLength ):
    # Choose random characters from this string:
    uppers = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    lowers = "abcdefghijklmnopqrstuvwxyz"
    others = "0123456789!@#$%^&*()_+={}[]|<>?/"
    choices = uppers + lowers + others
    numChoices = len( choices )

    # String together random characters until
    # desired length
    password = ""
    for i in range( passwordLength ):
        ch = choices[ random.randrange( 0, numChoices ) ]
        password += ch
    return password
```

Generating Random Passwords

```
def main():
    password1 = generatePassword( 10 )
    print("Password1: ", password1 )

    password2 = generatePassword( 20 )
    print("Password2: ", password2 )

    password3 = generatePassword( 30 )
    print("Password3: ", password3 )
    print("No one's going to break that one!")

main()
```

```
> python Password2.py
Password1:  s=)LR5gt50
Password2:  P^&27tO5!mk4y+>]o$>]
Password3:  EVsZj>iW=7??_gr?_>Ge9^_YQrK6md
No one's going to break that one!
```