Subprograms

Make our snowflake program use a function that draws one snowflake point.

Make our rhombus flower program use a function that draws one rhombus.

As your programs become larger, managing all of the details starts to become a real problem. One of the tools that programmers use to manage complexity is one that we have already used – the subprogram. A subprogram is exactly what it sounds like: a little part of a program.

**Subprogram:** a set of instructions that is given a name and can be "called" (executed) as needed.

Subprograms are also called *functions, procedures*, and *methods*.

In Python, they are usually referred to as *functions*.

## Functions

A function is a collection of instructions that is given a name. A function should perform a single task, like initializing the variables in a program, or loading the program's content. If you have two distinct tasks to perform, you probably need two separate functions. Writing functions that are focused like this allows the programmer to narrow his focus and concentrate on solving a specific part of the overall problem. This helps to reduce the complexity of the overall program, which makes the overall program easier to write.

A function's instructions get executed when they are "called". To call a function in Python, simply write its name, followed by an argument list enclosed in parentheses.

We have been using functions since day #1. *print* is a function. *int* is a function. *float* is a function.

print("Hello!")

The "Please enter your name:" string is information that we are *passing* to the function and is called an *argument*. An argument is a data item that is used to send information to a function. Sometimes functions need to have some information passed to them before they can do their job. In those cases, we are to provide this data in the form of an *argument list*.

## Creating a new function

To create a new function:

def <functionName> (<argumentList>):

<statements>

You can put any Python statements that you want after the function header. Unless they are controlled by other statements (like the body of a *for* loop), they must all be indented the same number of spaces (4 seems to be the accepted convention in Python).

#### Example:

Create a Python program called *Hello*. Write this function. This function prints the message "Hello" in the output window.

def greeting():

print("Hello!")

Add code to call the function in the program's main procedure:

greeting()

Once a function has been written, it can be called (executed) many times:

greeting()

greeting()

greeting()

Here's another way to call the function multiple times:

for i in range(10):

greeting()

# Arguments/Parameters

Functions can receive parameters from the caller.

#### Example

The *greeting* function could be personalized by passing in an argument giving a name to say hello to. Create a second function called *greeting2*:

def greeting2 (name="buddy"):

print("Hello, " + name + "!")

Note that when we use parameters, we can give the parameter a default value. Then, if the function is called without that particular parameter, the default will be used. So if we call the function like this:

greeting2()

the name "buddy" will be printed out.

IF we want to pass in a name, we do so like this:

greeting2 ("George")

greeting2 ("Martha")

And, instead of building the names into the program, we could get names from the user. Add the following to your main program:

theName = input("Enter a name: ")

greeting2 (theName)

theName = input("Enter a name: ")

greeting2 (theName)

It is also legal to pass more than one argument to a function. Write a new function called *greeting3:*

def greeting3 (name="buddy", count=1):

for i in range(count):

print("Hello " + name + "!")

Call it:

greeting3("Bob", 5)

greeting3("Chuck")

greeting3()

Note that the order of the arguments matters. And, if you omit arguments, they must be omitted from the end of the list. For example, you cannot do this:

greeting3(,5)

#### Why use subprograms?

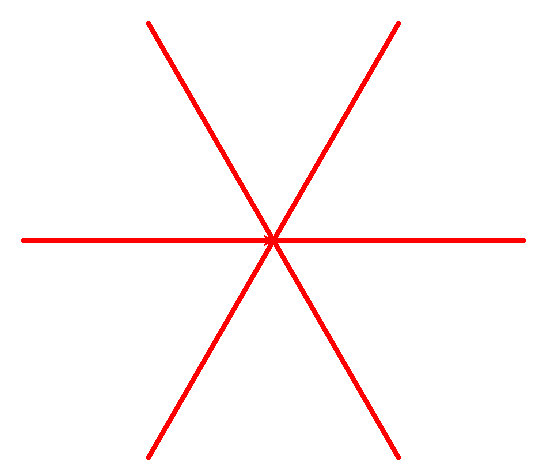
There are two very good reasons for using functions:

1. Functions let you divide your code into modules
2. Functions let you re-use code.

Dividing your code into modules makes it easier to organize your program, and if it is easier to organize your program, it will also be easier to understand and write your program. When you write a subprogram, you are writing a "little program" and you can focus on the task at hand. Small problems are easier to solve than big problems. If you break a big program up into a series of small programs and then focus on solving each small program one at a time, it allows you to focus on that part of the program. It *reduces the amount of information that you have to keep track of*. When you have finished one subprogram, you can essentially forget about it and move on to the next subprogram, only focusing on the details of that subprogram, etc., until you have eventually solved the big problem by solving a series of smaller problems. Imagine trying to solve a problem where you had 100 things to keep track of all at once. Now imagine trying to solve 10 small problems where you have to keep of 10 things when solving each small problem. The second approach is far easier than the first. As the problems you try to solve become larger, this "divide-and-conquer" approach becomes more and more useful.

# A star function

Let's write the code to draw a simple star, like this:



Ask the user for the number of points (int), color (string), and length of each point(int).

Now put that code in a function.

Do we want a 5-pointed star every time? No, so make the number of points a parameter.

Do we want the same size star every time? No, so make the side length a parameter.

Do we want the same color every time? No, so make the color a parameter.

We should also pass in the turtle that we are using to draw the star. Here is the first line of the function definition:

def star(t, points=5, size=200, color="red"):

t.color(color)

t.pendown()

angle = 360 / points

for i in range(points):

t.forward(size)

t.back(size)

t.right(angle)

t.penup()

# More Turtle functions

What if we wanted to draw triangles, squares, pentagons? Let's write some functions.

def triangle(t, side=200, color="red"):

t.color(color)

t.pendown()

for i in range(3):

t.forward(side)

t.left(120)

t.penup()

def square(t, side=200, color="red"):

t.color(color)

t.pendown()

for i in range(4):

t.forward(side)

t.left(90)

t.penup()

def pentagon(t, side=200, color="red"):

t.color(color)

t.pendown()

for i in range(5):

t.forward(side)

t.left(72)

t.penup()

Note that all three functions are almost identical. The only differences are in the range, and in the command to turn left. The range argument is always the number of sides, and the turn left argument is always 360 divided by the number of sides. So we really don't need three functions! We can write one *polygon* function where we pass in one more argument: the number of sides.

def polygon(t, sides=3, side=200, color="red"):

t.color(color)

t.pendown()

for i in range(sides):

t.forward(side)

t.left(360/sides)

t.penup()

t.speed(10)

for i in range (3,11): # polygons from 3 sides to 10 sides

t.penup()

t.goto(-100, -300)

polygon(t, i)

Now we can, with one function, create a polygon with any number of sides, any size, and any color.

# Yesterday's problems: A draw line function

Last class we were using nested loops to draw patterns on the screen. This is a great place to use functions.

One thing that we had to do repeatedly was print a bunch of stars, so we could write a *printStars* function. The only thing it needs to know is the number of stars to print.

def drawStars(count):

Write the body.

We also needed to print a bunch of blanks.

def printBlanks(count):

Write the body.

Note how similar both functions are!

We can replace them with a single function that accepts TWO arguments: the character to print, and the number of times to print it. Note that we will have to change the name of the function to something more general.

def printChars(char, count):

Write the body.

Functions that return values

It is also possible to write functions that return a value to the caller. Python has a number of built-in functions that return a value.

#### Examples

The absolute value function, *abs*:

print(abs(-5))

The maximum function, *max*:

print(max(1, 2, 3))

print(max(3 \* 11, 5 \*\* 3, 512 - 9, 1024 \*\* 0))

# Writing our own functions that return a value

Write a function to compute the square of a number:

def square(x=0):

y = x \* x

return y

We can call the function like this:

print(square(10))

Write a function to raise a number to a power:

def power(base=10, exponent=1)

result = base\*\*exponent

return result

Write a function to find the area of a rectangle:

def area(length=0, width=0):

return length \* width

Write a function to find the area of a circle:

def circleArea (radius=0)

return radius \* radius \* 3.14159

Instead of using 3.14159, you could use *math.pi*, but that requires importing the *math* module:

import math

def circleArea (radius=0)

return radius \* radius \* math.pi