CIS 231

Python – Ch. 3

Functions

(+ 4.2 – Simple repetition)

3.1 Function Calls

- Type the name of the function
  - May also pass one or more *arguments*
- Arguments are the information you are required to pass to the function (*parameters*)
- Functions can also have a *return value*, which is covered in a later chapter

Type Conversion Functions

- Similar to casts (type casts) in other languages
- Converts a copy of the data to a different type
  - `float` to `int` results in *truncation*
    - (only whole number portion survives)
- Can convert to string with `str()`
  - becomes a set of characters rather than a numerical value

3.2 Math Functions

- Handle well-known math operations
- Part of the math *module*
  - A module is a set of related functions
  - Make available by adding `import math`
- Includes functions such as `sqrt()`, `log10()`, and trig functions
- We typically use `()` when referring to a function to distinguish it from a variable

3.3 Composition

- Arguments to functions may be *composed* of the result of multiple functions and operations
  - The result is computed and sent to the function

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
```

- You can also pass the result from a call to one function as an argument to another function:

```
x = math.exp(math.log(x+1))
```

  - Note nested parentheses

3.4 Adding new functions

- You can define new functions just like you can define new variables
- A function is composed of a collection of statements
- Start with `def` and the function name
  - Then add the statements for the function
  - Type an empty line to end the definition
- The first line is known as the *function header*

3.5 Definitions and uses

- Defining a function doesn't guarantee that it will be executed
- Defined functions need to be called just like the built-in functions
- Functions can include calls to other functions

3.6 Flow of execution

- The order in which statements are executed
- Done in linear order of statements
  - Until we start conditional execution (Ch. 5)
- Calling a function pauses execution in the calling function and transfers it to the function being called – when it returns execution resumes in the calling function
- Review code by following the flow

3.7 Parameters and arguments

- Functions can define variables that receive the information passed to it – these are known as the *parameters* of the function
- When calling that function – the caller is expected to provide an *argument* for each parameter the function requires
- Each parameter is assigned the corresponding argument that is passed to it
- Parameters are defined by the function, arguments are specified for each function call

3.8 Variables/parameters are local

- Local is a term that describes the *scope* of a variable
- Scope dictates a variable's visibility and lifetime
- A local variable is only seen (accessible) in the scope where it is declared
- A local variable is destroyed when the flow of execution leaves the scope where it was declared ("goes out of scope")

3.9 – Stack diagrams

- Helpful in showing the flow between functions
- When functions are called, the function that called it is placed on top of a *stack*
- Stacks are known as a LIFO (Last In, First Out) data structure
- When leaving a function we return the one that called it, which will be on the top of the stack
- This stack can perform a *traceback* (stack trace) when there is an error

3.10 Fruitful/void functions

- In the author's parlance, functions either return a value ("fruitful") or they don't ("void")
- The math functions we've encountered do return values, which can be used for assignment or in expressions
- We're writing void functions for now, but that will change soon

3.11 Why functions?

- They may not seem necessary now, but they make it a lot easier to build larger programs
- Allow you to write code that is specific to a task, not specific variables
- Break your program into smaller pieces, making it easier to write, test, and debug
- Make it easier to reuse code you've written in future programs

4.2 Simple repetition

- Also known as *iteration*, or *looping*
- The for loop is the most common
  - Designed to repeat a set of statements a particular number of times (variable or literal)

```
for i in range(4):
    print (i)
```

  - `i` takes on the values 0 to 3, and does one iteration of the loop for each value of `i`
  - Note indention of code in loop

Up Next

- Ch. 5 – Conditionals and Recursion
  - Modulus operator
  - Boolean (T/F) expressions
  - Logical operators
  - Conditional execution
  - Alternative execution
  - Chained/nested conditionals
  - Recursion