Defining Functions and Modules

CSE/IT 107L

NMT Department of Computer Science and Engineering

"How do we convince people that in programming simplicity and clarity – in short: what mathematicians call elegance – are not a dispensable luxury, but a crucial matter that decides between success and failure?"

— Edsger Dijkstra

"Simplicity is the ultimate sophistication."

— Leonardo Da Vinci

"Only ugly languages become popular. Python is the exception."

— Donald Knuth

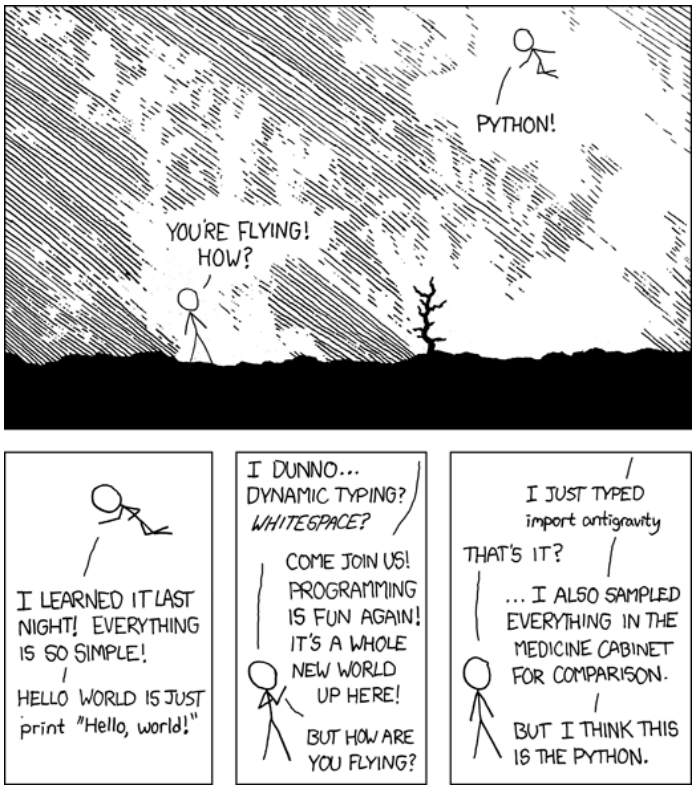


Figure 1: `http://xkcd.com/353`

Introduction

This lab is meant to introduce the `"".format()` function for printing values, a new type of loop called the `while` loop, and the reasons and methods for defining Python functions and modules.

Contents

1 Convert Variables and Values to Formatted Strings Using `"".format()`

Previously, when we wanted to print out both a number and a string, we had to print them on separate lines:

```
print("x is equal to:")
print(x)
```

However, there is a more effective way to print Python values. The `"".format()` method offers many more options for formatting output:

```
>>> x = 5
>>> print("x is equal to: {}".format(x))
x is equal to: 5
>>> import math
>>> print("5 digits of pi: {:.5f}".format(math.pi))
5 digits of pi: 3.14159
```

In the first example, the function replaces the `"{}"` in the string with the value of `x`. If we include multiple instances of `{}` in our string, we are able to pass multiple values to `"".format()`. It will place each of the values into the string.

```
>>> x = 5
>>> y = 6
>>> print("x is equal to {} and y is equal to {}.".format(x, y))
x is equal to 5 and y is equal to 6.
>>> print("row {}: {},{},{}".format(1, x, y, 1))
row 1: 5,6,1
```

1.1 Printing Numbers

We can use the `"".format()` method to print a specific number of decimal places. To do this, add `:.2f` inside of the `{}`. The `.2f` specifies that we want 2 digits to follow the decimal point.

```
>>> print(3.141592653589793)
3.141592653589793
>>> print("{:.2f}".format(3.141592653589793))
3.14
```

For more powerful formatting options, see

`https://pyformat.info/`

`https://docs.python.org/3/library/string.html#format-string-syntax`

1.2 Alternatives

You are encouraged to use `"".format()` in your exercises, but there are several other ways of achieving the same results. For the sake of simplicity and brevity we will only show brief demonstrations. If `y = 5`, then the lines `print('X ' + str(y) + ' Z')`, `print('X %s Z' % y)`, and `print("X", y, "Z")` have the same effect.

2 Repetition with `while` Loops

As we saw in lab 0, Python's `for` loops have limitations. It is not possible to use them to repeat code for an indefinite number of times. To do that, we'll have to use `while` loops, which allow for more free-form iteration. The syntax of a `while` loop is very similar to that of an `if` statement, but instead of only running the indented block of code once, the loop will continue running it until the given boolean statement is no longer true. In general, `while` loops look like:

```
while boolean_condition:
    # run this code until the boolean is False
    pass
# run this code once the while loop finishes
```

The special keyword `pass` is a placeholder for indented Python code. Here's an example:

```
x = 10
while x > 0:
    print(x)
    x = x - 1
```

The above program will print out the numbers 10 to 1. There are many ways to decide when a while-loop should stop running its code — the previous demonstration involves variable reassignment. Every time the loop runs, the code assigns a smaller value to the Python variable x. Eventually, the variable is smaller than 1 and the loop stops running. It is similar to this code:

```
x = 10
if x > 0:
    print(x)
    x = x - 1
    if x > 0:
        print(x)
        x = x - 1
        if x > 0:
            ...  # arbitrary depth!!!
```

Try to run the following code manually. What should it print?

```
x = 10
while x > 0:
```

```
    x = x - 1
    print(x)
```

This version of the program will print out the numbers 9 to 0. This might seem a bit strange, since the condition of the loop says it will stop when x is no longer larger than 0. And yet, it prints out the value 0 before the loop ends. This is because the loop condition is only checked whenever the end of the indented section is reached. If the condition is `True`, then the indented section will be executed again. If the condition is `False`, then the loop will end.

If the boolean value starts out `False`, then the loop will never execute. An example:

```
x = 0
while x > 0:
    x = x - 1
    print(x)
```

2.1 Infinite Loops

Although while-loops are more flexible, they are also more **dangerous** and harder to debug than for-loops. If the boolean statement is never `False`, the Python code could go into an *infinite loop*.

```
while True:
    print("Printing forever")
```

Press *Ctrl+C* to stop this loop.

2.2 Indefinite Loops

Let's write a clone of the Linux command `cat`. It should take a line of the user's input and immediately print it.

```
user_input = ""  # no input yet...
while user_input != "exit":
    user_input = input()
    print(user_input)
```

The `while` loop depends on user input. It only stops when the user types "exit." Otherwise, it keeps prompting for more input. This loop runs for an indefinite, maybe even infinite, number of times.

For example, the Python interactive shell reads lines of Python code until you type Ctrl+D or "exit". Video games draw frames to the screen until the graphics engine needs to shutdown. Many other programs need to loop for an indefinite number of times.

2.3 Reading User Input

The following sample program repeatedly prompts for user input and prints the sum of all inputs until the user types either "exit" or "forget". This is also an example of conditional statements nested within `while` loops. There could also be a while-loop within a conditional statement.

```
user_input = ""
sum = 0
print("Type 'exit' to quit, or 'forget' to reset")

# stop when the user's input is 'exit'
while user_input != "exit":
    user_input = input("Please type a number: ")
    if user_input == "forget":
        sum = 0
    elif user_input == "exit":
        print("goodbye!")
    else:
        user_input = int(user_input)
        sum += user_input
        print(sum)
```

3 Defining Functions

As we have seen, functions can be called with a specific number of arguments and some functions return values that can be assigned to variables. For example, the `print(x)` function takes an argument and returns nothing; `y = input(x)` takes an argument and returns a string; and the `turtle.goto(x, y)` function takes two arguments and returns nothing. In this section, we will describe the idea of a function and you will learn how to write your own functions.

3.1 Intuition

A function accepts some number of parameters as input, it could also take no input. A function runs code using its inputs, and then returns an output. A function should have a clearly defined purpose and a short but descriptive name. Think of a function as a subprogram or a machine that is available for use by any part of the Python program.

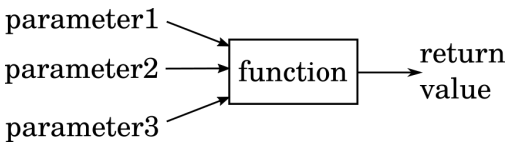


Figure 2: A diagram of a function. Inputs are on the left, output is on the right.

Functions are important because they are:

Modular Functions allow us to break our programs into many smaller pieces. This also allows us to easily think about each small piece in detail.

Easy to Test Functions allow us to test small parts of our programs while not affecting other parts of the program — this reduces errors in our code.

Reusable Instead of writing the same code many times, we can place this code within a function and call the function whenever that code is needed.

3.2 A Function With No Input or Output

The simplest function is one that takes no input and returns no output. It is defined by writing the `def` keyword, the function's name, matching parantheses, and a colon. Then, the code that should run when the function is called must be written and indented. Some examples of these functions are the `turtle.done()` and the `exit()` functions.

```
def function_name():
    # Python code
    pass
# to run the Python code:
function_name()
```

When called, the following function, prints "goodbye" and then exits the program:

```
>>> def goodbye_and_exit():
...     print("Goodbye.")
...     exit()
>>> goodbye_and_exit()
Goodbye.
```

3.3 A Function With Input and No Output

A function can take zero or more parameters as input. For example, the `print()` function takes a Python value as its input. To write a function that takes parameters, write them within the parentheses. The parameters can be used from within the function's code as though they are variables.

```
def function_name(parameter):
    # Python code uses the variable named `parameter`
    pass

def function_name2(parameter1, parameter2, parameter3):
    # Python code that uses all 3 parameters
    pass

# Using these functions:
function_name("sample")  # `parameter` has value set to "sample"
function_name2(5, 9, 3)  # `parameter1` is set to 5
```

```
>>> import turtle
>>> def draw_hexagon(side_length):
...     for i in range(6):
...         turtle.forward(side_length)
...         turtle.left(360 / 6)
```

3.4 A Function That Produces Output Using the `return` Keyword

When we used functions from the math module, we were always able to assign the result of a function to a variable. For example:

```
>>> import math
>>> x = math.sqrt(16)
>>> print(x)
4.0
```

So how do we get a function to give back a value — or *return* a value? We write a return statement at the end of the function by writing the `return` keyword and the Python value or variable that the function should output.

```
def function_name(parameter1, parameter2):
    # Python code uses variables named `parameter1`, `parameter2`
    # defines `result` somewhere
    result = ...
    return result

# To use it:
x = function_name(2, 10)  # parameter1 set to 2, x set to return value
```

An example:

```
>>> def square(x):
...     return x ** 2
...
>>> y = square(5)
>>> print(y)
25
>>> square(4.3)
18.49
```

As soon as a `return` statement is reached, the function stops executing and just returns the value given to it. Any subsequent statements that are part of the function will be skipped.

If a function is called with a different number of parameters than it was designed for, Python will raise a `TypeError` with a message that says how many parameters it should take. This function takes two parameters and returns a numeric value:

```
>>> def wage(hours, base_rate):
...     if hours > 40:
...         ot_pay = (hours - 40) * base_rate * 1.5
...         return base_rate * 40 + ot_pay
...     pay = hours * base_rate
...     return pay
...
>>> wage(40, 10)
400
>>> wage(50, 10)
550.0
>>> wage(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    wage(10)
TypeError: wage() missing 1 required positional argument: 'base_rate'
>>> wage(10, 20, 30)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    wage(10, 20, 30)
TypeError: wage() takes 2 positional arguments but 3 were given
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    wage(10, 20, 30)
TypeError: wage() takes 2 positional arguments but 3 were given
```

The `grocer` function takes two parameters and returns a string:

```
>>> def grocer(num_fruits, fruit_kind):
...     return 'Stock: {} cases of {}'.format(num_fruits, fruit_kind)
...
>>> grocer(37, 'kale')
'Stock: 37 cases of kale'
>>> print(grocer(0, 'bananas'))
Stock: 0 cases of bananas
```

A function may also call other functions. Here is the wage example, but now the `wage_after_tax` function uses the `wage` function:

```
def wage(hours, base_rate):
    """Calculate and return weekly pay for a given amount of hours and base rate taking
    into consideration overtime pay at 1.5 times the given rate."""
    if hours > 40:
        ot_pay = (hours - 40) * base_rate * 1.5
        return base_rate * 40 + ot_pay
    pay = hours * base_rate
    return pay

def wage_after_tax(hours, base_rate, tax_rate):
```

```
    """Calculate and return weekly pay after taxes for a given amount of hours and a
    base rate with a flat tax rate."""
    pay = wage(hours, base_rate)  # use the previously defined function
    return pay * (1 - tax_rate)
```

A good resource:

`https://docs.python.org/3/tutorial/controlflow.html#defining-functions`

4 Reusing Code: Creation and Usage of Modules

When writing large programs, it is convenient to split them into small files. This makes it easier to test and debug small parts of the entire program. How can a programmer use code from two or more different `.py` files?

The solution is to place your code into modules. A module is essentially a `.py` file. In fact, you've been writing Python modules this entire semester. You could import the file `test.py` and access anything defined within it by writing `import test`. However, these files are likely are missing one bit of code to work properly.

4.1 Bad Solution: Copy and Paste

You could copy and paste the code you wish to share between two `.py` files, but this is a terrible idea under almost any circumstance. Any fixes or improvements to either segment of shared code must be copied manually. If the code is used frequently, this might have to happen multiple times.

If you use modules, a programmer can immediately tell that you're using code from another file and understand that they may need to interact with the imported file as well.

4.2 Half Solution: Import Your Code

For learning purposes, let's try to write and import a broken module. Place the following text in a file called `bad_artifical_intelligence.py`.

```
def greetings(recipient):
    greeting = "Hello {}!".format(recipient)
    return greeting

# test the greetings functions
if greetings("World") == "Hello World!":
    result = "passed"
else:
    result = "failed"
print("The `artificial_intelligence` module has {} all tests!".format(result))
```

Here we have a function called `greetings` that takes a string and returns a string. After the function is defined, we test it by giving it the input `"World"` and checking that it outputs the correct string,

`"Hello World!"`. If we run `python3 bad_artificial_intelligence.py` on its own, it will print the expected output:

```
The `artificial_intelligence` module has passed all tests!
```

However, this is module still **broken**! If we wish to use the function `greetings` in some other Python program, then we'll see that it has unexpected side effects. Now let's see an example where we `import` our existing code and use it in another program, called `greeter.py`:

```
import bad_artificial_intelligence

user_name = input("Please type your name: ")
greetings = bad_artificial_intelligence.greetings(user_name)
print(greetings)
```

Side Note: To import another module, you need to make sure that both files are in the same directory! Otherwise, you will see this error message:

```
$ python3 greeter.py  # files in wrong directories
Traceback (most recent call last):
  File "greeter.py", line 1, in <module>
    import bad_artificial_intelligence
ImportError: No module named 'bad_artificial_intelligence'
```

Here's what happens when `greeter.py` is run correctly:

```
$ ls
greeter.py  bad_artificial_intelligence.py

$ python3 greeter.py
Please type your name: gazorpazorpfield
The `bad_artificial_intelligence` module has passed all tests!
Hello gazorpazorpfield!
```

The problem here is that `greeter.py` is also printing the output of `bad_artificial_intelligence.py`. This isn't what we want! The testing code in our module has nothing to do with our greeter printing script! How can we make our module more *modular*? In this case, the problem is in how we wrote the `bad_artificial_intelligence.py` file.

4.3 Full Solution: Import Your Code and Check the `__name__` Variable

Please refer to the previous section if you haven't written the file `bad_artificial_intelligence.py`.

The problem with the previous version of the file `greeter.py` is that the testing code for `bad_artificial_intelligence` was always running, even when the module was being used only for its `bad_artificial_intelligence.greetings` function.

The `math` and the `turtle` modules are defined in the `math.py` and `turtle.py` files located somewhere on your machine. They may have testing code, or sample code, but this code would only run if you were to type `python3 /crazy/file/path/turtle.py` or `python3 -m turtle` (try this for different modules).

The question is: how does a file know whether it's being imported as a module using `import test_file` or if it's being run as a program using `python3 test_file.py`? Python keeps a variable called `__name__`. It is set to the file name, unless the program is being run using `python3 test_file.py`. In this case, the variable is set to `"__main__"`.

Here is the corrected version of the `bad_artificial_intelligence.py` file, now called `artificial_intelligence`:

```
def greetings(recipient):
    greeting = "Hello {}!".format(recipient)
    return greeting

# If this file is being run as `python3 artificial_intelligence`, run some testing code
if __name__ == "__main__":
    if greetings("World") == "Hello World!":
        result = "passed"
    else:
        result = "failed"
    print("The `artificial_intelligence` module has {} all tests!".format(result))
```

Change the `greeter.py` Python script so that it imports and uses this module instead. Its output:

```
$ python3 greeter.py
Please type your name: gazorpazorpfield
Hello gazorpazorpfield!
```

4.4 Boilerplate Code: Check if `__name__ == "__main__"`

All of your programs are required to have the boilerplate code for checking if `__name__` equals `"__main__"` and a `main()` function for running code that doesn't belong in any other function.

Any code that previously would have gone outside of any function declaration should now go inside a `main()` function as shown below. This is so you can test your `main` code from the Python interactive shell, or from another program.

Furthermore, it's strongly advised that all of the code in your program that uses IO functions, like `input()` or `print()`, should only be called from within the `main` function! This is the best way to avoid the situation we ran into in our example, where our module was printing messages unrelated to the program that was using it. When you're reusing code from some other program, you don't want the functions you use to clutter up your program by printing irrelevant messages to your screen.

This code follows the boilerplate requirement:

```
def f(x):
    return x % 5

def main():
    # test and demo f
    print(f(5) == f(10))
    i = input()
    i = int(i)
    print(f(i))

if __name__ == "__main__":
    main()
```

This code is doing it wrong:

```
def f(x):
    print("sup br0s?")
    print("I just got {}!".format(x))
    print("Think it's divisible by 5?")
    return x % 5


# test f
print(f(5) == f(10))
i = input()
i = int(i)
print(f(i))
```

Be the change you wish to see in the world. Write clean code.

5 Making Calculations Shorter

Python operators such as +, -, *, %, were introduced in Lab 1. There is a variant of these that you can use to assign to a variable.

```
>>> x = 5
>>> x += 3 # same as x = x + 3
>>> x
8
>>> x *= 10 # same as x = x * 10
>>> x
80
```

The available assignment operators are:

+= addition	*= multiplication	//= integer division
-= subtraction	/= division	**= exponentiation

6 Sample Program

This sample program defines some relatively short functions and a `main()` function for getting input and printing the return value of both functions.

```
import math

def first_root(a, b, c):
    root = -b + math.sqrt(b ** 2 + 4 * a * c) / (2 * a)
    return root

def second_root(a, b, c):
    root = -b - math.sqrt(b ** 2 + 4 * a * c) / (2 * a)
    return root

def main():
    # Get three floating point numbers as input until the user types 'exit'.
    print('Type the coefficients of a quadratic equation a*x**2 + b*x + c=0.')
    print('Type "exit" to finish.')
```

```
15
16      user_input = 'y'
17      while user_input == 'y':
18          a = float(input('a > '))
19          b = float(input('b > '))
20          c = float(input('c > '))
21
22          # Calculate and print the roots.
23          root1 = first_root(a, b, c)
24          root2 = second_root(a, b, c)
25          if root1 == root2:
26              print('Repeated root is {}.'.format(root1))
27          else:
28              print('Roots are {} and {}.'.format(root1, root2))
29
30          user_input = input('More input? Type (y) or (n) > ')
31
32  if __name__ == '__main__':
33      main()
```

7 Exercises

New Requirements

Please be aware of the new code style requirement. See lab 2 for a description of PEP 8.

Also, we now require your programs to have the boilerplate code as shown in Section 4.4 on page 10.

Exercise 7.1 (fizzbuzz.py).

Have the user enter a positive integer number. Then, print the numbers from 1 to that number each on a line. When the printed number is divisible by 3, print "Fizz", and when the number is divisible by 5, print "Buzz", and when it is divisible by both, print "FizzBuzz".

You must use `.format()` and a `while` loop.

Should look like this when run:

```
Enter a number: -1
Not a positive number!
```

```
Enter a number: 16
1
2
3 Fizz
4
5 Buzz
6 Fizz
7
8
9 Fizz
10 Buzz
11
12 Fizz
13
14
15 FizzBuzz
16
```

Exercise 7.2 (date.py, horoscope.py).

For this exercise you will need to write two different Python modules. Make sure you follow the boilerplate requirements for both programs, so that the two modules don't interfere with each other.

The first program will be in `date.py`. It should ask the user for a month and a day of the month, and convert it to the corresponding day of the year. Assume the current year is not a leap year. For incorrect dates, like `('February', 29)`, `('March', 42)` or `('Scotchtober', 1)`, your conversion function should return `-1`, and you should print an error from the `main` function.

Hint: Both of these programs will be easier to write if you write a helper function, `days_in_month()`, which takes in the name of a month and returns the number of days in that

month.

Example Input:

```
Enter the month: March
Enter the day: 14
```

Output:

```
Day of the year: 73
```

Next, `horoscope.py` should take in a month and a day, and use that date to print the horoscope of someone who was born on that day. You should use the `import` statement to make use of the code you wrote in `date.py` for determining the user's astrological sign. The horoscopes themselves are completely up to you, as long as they start with the correct sign.

Example Input:

```
Enter the month: March
Enter the day: 14
```

Output:

```
Pisces: Tui and La, push and pull, commit and rebase...
```

For a reference on the zodiac dates, see the "Tropical zodiac" column from this table: `https://en.wikipedia.org/wiki/Zodiac#Table_of_dates`. Watch out for Capricorn! That zodiac spans from December 22nd to January 20th.

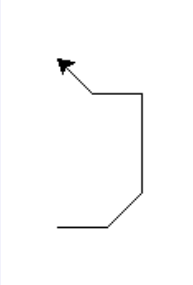
Exercise 7.3 (navigate.py).

Write a program that takes directions from the command line to draw a line. Let the user input "left", "right", "forward", or "stop". Left and right turn the turtle left or right however many degrees are entered, forward moves the turtle forward (however far you wish), and stop ends the program.

Input:

```
Please enter a direction: forward
Please enter a direction: left
How many degrees? 45
Please enter a direction: forward
Please enter a direction: left
How many degrees? -1
Invalid number, not moving.
Please enter a direction: left
How many degrees? 45
Please enter a direction: forward
Please enter a direction: forward
Please enter a direction: left
How many degrees? 45
Please enter a direction: left
How many degrees? 45
Please enter a direction: forward
Please enter a direction: right
How many degrees? 45
Please enter a direction: forward
Please enter a direction: stop
```

Output:



Index of New Functions and Methods

Submitting

You should submit your code as a tarball that contains all the exercise files for this lab. The submitted file should be named

`cse107_firstname_lastname_lab3.tar.gz`

Upload your tarball to Canvas.

List of Files to Submit

Exercises start on page 13.