UNIVERSITY OF WATERLOO

Lesson 1.3: Functions

In this lesson, we discuss calling and defining functions in Python. Python comes equipped with a number of built-in functions. Some of the more common ones we will use include the following:

- `abs(n)` returns the absolute value of the integer or floating point number `n`
- `len(s)` consumes a string `s` and returns the number of characters in `s`
- `max(arg1, arg2, ...)` returns the maximum of the arguments passed (note the arguments must be types that can be compared).
- `min(arg1, arg2, ...)` returns the minimum of the arguments passed (note the arguments must be types that can be compared).

The last two functions are interesting in that they are functions that can take in a variable number of parameters! This is a very powerful feature of Python — the ability to write functions that can consume a non-fixed number of arguments. We won't define such functions ourselves but note these are very useful!

What follows are examples of how to call these functions. Notice that in Python, we say that a function **returns** a value instead of **produces** a value. We will always use return to describe what a function output is and this will be reflected in our design recipe at the end of this module.

- `abs(-3.8)` returns `3.8`.
- `len("Hello There")` returns `11`.
- `max(3, 5.2, 9)` returns `9`.
- `min("ABC", "AA")` returns `"AA"`.

Notice above that we can call `max` and `min` with strings as well! The ordering used on strings is that as described by the ASCII table, which we speak about much later in the course. For now, note the order is natural and numbers precede words beginning with capital letters and these precede words beginning with lower case letters. As an example, `max('0ab', 'ZZZ', 'apple')` is `'apple'`. In general, to call a function in Python, we write:

```
fn_name(arg1, arg2, ..., argN)
```

This `fn_name` can either be a built-in function or a user-defined function. However, you must have the correct number of arguments and these arguments have to be separated by commas.

Concept Check 1.3.1

1 point possible (graded)
Consider the following piece of code:

```
n = max(len("spaces are fun!"), abs(-6.5), min(100, -100))
```

What is the value of `n`?

User-Defined Functions

In Python, you can create your own functions as well! The following gives the template for creating a function with `N` parameters:

```
def fname (p1, p2, ..., pN):
  statement1
  statement2
  ...
  statementK
```

There are lots of important characteristics to creating a user-defined function and many of these are subtle:

- The declaration begins with the keyword `def` followed by a valid function name `fname` and finishing with a parenthetical argument list and a colon.
- If you type the declaration correctly and press enter after the colon, our text editor should automatically **indent** the first line appropriately. This indentation is important — indentation is how Python determines which lines of code belong to the body of the function. Contrast this to other imperative languages that use braces to keep track of this. The indentation can be whatever spacing you want but must be consistent. Typically, we recommend either 2 or 4 spaces throughout your code. We will use 2 spaces to maximize how much information we can put in a line. Also note that tabs and spaces are different characters (which can also cause issues for students). In our editor, the tab key will print spaces but if you copy and paste code form elsewhere, this can be a common source of error.
- Lastly, the body of the function continues to the end. Execution stops when either the end of the body is reached or, more likely, when the function executes a `return` statement. This statement tells Python to stop the function and return the value corresponding to the experssion after the keyword `return`. If no return statement is present, the function returns a default value of `None`. This might seem odd at first but in a few modules you will see why this might actually be useful. As a consequence of this, all functions in Python have a return value! Note that arithmetic operations combined with `None` will give an error. This value `None` is added to our style guide as a possible data type a function can consume or return.

Note we have the first instances of keywords in Python, namely `def`, `return` and `None`. Note that in Python we are **not** allowed to have variable names equal to these keyword names. It should not be hard to create variable names that avoid this restriction! Further, function names in Python follow similar rules to variable names. For us, we will always use snake case in assignments and encourage students to do the same (that is, start with a lower case letter for function names and separate words with underscores).

Let's try to create some of our own functions!

Concept Check 1.3.2

1 point possible (graded)
Which of the following code expressions always evaluates to the last digit of a natural number `n` for any such natural number? It might help to try the following with multiple sample values of `n`

- ○ `n % 10`
- ○ `n // 10`
- ○ `n * 10`

○ `n % 100`

Concept Check 1.3.3

Write a function `last_digit(n)` that returns the last digit of `n`. Accomplish this by starting line 2 with two spaces, then writing `return` and then the answer from the previous concept check.

0/3 points
Attempts: 0 / Unlimited

```
def last_digit(n):
```

Concept Check 1.3.4

1 point possible (graded)
Which of the following code expressions always evaluates to the tens digit of a two digit natural number `n`? It might help to try the following with multiple sample values of `n`

○ `n % 10`

○ `n // 10`

○ `n * 10`

○ `n % 100`

Sum Digits of a Two Digit Natural Number

In what follows, we write a function that produces the sum of the digits of a two digit number. At the end of the code, we also give an example of calling the code by setting `d = sum_digits(74)` so that you can see what happens with the visualizer.

As an extension to think about, how would you accomplish this for a three digit number using the ideas of calculating `n % 10` and `n // 10`?

```
def sum_digits(two_digit_number):
    ones = two_digit_number % 10
    tens = two_digit_number // 10
    total = ones + tens
    return total

d = sum_digits(74)
```

Visualize Reset Code

Common Errors

It is extremely easy to make a typo when writing these functions for the first time. Extremely common mistakes include:

- forgetting `return` statements
- forgetting the keyword `def`
- missing a colon at the end of a function declaration
- indentation errors

See if you can debug each of the following attempts to code `sum_digits`. Then, try to debug the final piece of code.

Concept Check 1.3.5

The following attempt of `sum_digits` contains an error. Find and fix the error so that the function is defined correctly and correctly returns the sum of the digits if given a two digit number.

0/3 points
Attempts: 0 / Unlimited

```
def sum_digits(two_digit_number):
    ones = two_digit_number % 10
    tens = two_digit_number // 10
    total = ones + tens
```

Concept Check 1.3.6

The following attempt of `sum_digits` contains an error. Find and fix the error so that the function is defined correctly and correctly returns the sum of the digits if given a two digit number.

0/3 points
Attempts: 0 / Unlimited

```
def sum_digits(two_digit_number):
    ones = two_digit_number % 10
    tens = two_digit_number // 10
    total = ones + tens
    return
```

Concept Check 1.3.7

The following attempt of `sum_digits` contains an error. Find and fix the error so that the function is defined correctly and correctly returns the sum of the digits if given a two digit number.

0/3 points
Attempts: 0 / Unlimited

```
sum_digits(two_digit_number):
    ones = two_digit_number % 10
    tens = two_digit_number // 10
    total = ones + tens
    return total
```

Concept Check 1.3.8

The following attempt of `sum_digits` contains an error. Find and fix the error so that the function is defined correctly and correctly returns the sum of the digits if given a two digit number.

0/3 points
Attempts: 0 / Unlimited

```
def sum_digits(two_digit_number)
    ones = two_digit_number % 10
    tens = two_digit_number // 10
    total = ones + tens
    return total
```

Concept Check 1.3.9

The following attempt of `sum_digits` contains an error. Find and fix the error so that the function is defined correctly and correctly returns the sum of the digits if given a two digit number.

0/3 points
Attempts: 0 / Unlimited

```
def sum_digits(two_digit_number):
    ones = two_digit_number % 10
       tens = two_digit_number // 10
           total = ones + tens
    return total
```

Concept Check 1.3.10

Write a function `middle(a, b, c)` that consumes three integers `a`, `b`, and `c` and returns the median of the three numbers.

Accomplish this by creating three variables. One is the `largest` of the three values using the built-in function `max`. One is the `smallest` using another built-in function and the `median` number is computed by taking the sum of the three parameters and removing `largest` and `smallest`.

0/3 points
Attempts: 0 / Unlimited

```
## WARNING!!
## This example contains indentation errors!

def tens_digit(n):
    '''returns the tens digit in n
       tens_digit: Nat -> Nat
```

```
          Examples:
          tens_digit(1234) => 3
          tens_digit(4) => 0
          '''
       div10 = n // 10
          tens = div10 % 10
     return tens
```

Visualize Reset Code

Reading Error Messages

When you make a mistake in Python, the error message you get might not match the one you want. Consider the following code. What is the error preventing the code from returning `secret` minus the sum of the digits of `secret`?

Adding None and Int

```
1   def sum_digits(secret):
2       ones = secret % 10
3       tens = secret // 10
4       total = ones + tens
5
6   def calculation(secret):
7       s = sum_digits(secret)
8       return secret - s
9
10  c = calculation(74)
```

The error for the code above is not something to the effect of "Missing `return` statement" because this is not an error in Python! Functions are allowed to not have return statements. The problem is if we are missing a return statement, our function returns `None` as stated above. Thus, the value of `s` is `None` and in line 8, we cannot perform the computation `secret - s` since you are not allowed to do arithmetic with `None`

Concept Check 1.3.11

1 point possible (graded)
Consider the following code:

```
def tens_digit(n):
  ans = (n // 10) % 10
d = tens_digit(123)
```

What is the value of `d` after the above code has been executed?

◯ 1

◯ 2

◯ 3

◯ None

◯ None of the above.

Concept Check 1.3.12

1 point possible (graded)
Consider the following code:

```
def bar(param):
  z = 4 + param
x = 3
y = bar(x) + 3
```

What is the value of `y` after the above code has been executed?

◯ 4

◯ 7

◯ 10

◯ None

◯ There is an error with the above code

Typecasting

Occasionally it will be convenient for us to convert between one type and another. Python gives us typecasting commands to accomplish this, namely `int()`, `float()`, and `str()`. Each consumes either an `Int`, `Float`, or a `Str` and returns a new value of the desired type. Try the following examples with the visualizer.

```
a = float(1)
b = float("34.1")
c = float("23")

d = int(4.7)
e = int(-4.7)
f = int(2)
g = int("1234")


h = str(123)
i = str(5.446)
j = str("string")
```

Visualize Reset Code

Notice however that the object being passed to these type casting functions must be valid and must be convertible. As examples, each of the following give an error:

```
float("2.7.2")
str(1.2.3)
int("1.3")
```

In the first case, `"2.7.2"` is not a valid floating point value. In the second, `1.2.3` is not a valid `Int`, `Float`, or `Str` value and so this will give an error. The third example is perhaps the trickiest but since `"1.3"` is not a valid integer, we also get an error. Notice however that `int(float("1.3"))` works correctly.

Lastly, there is also a built-in function that returns the type of a variable or an expression, `type()`. Try this with the visualizer below.

```
x = type(1)
y = type(str(1.2))
z = type("hot" + "dog")
```

Visualize Reset Code

Concept Check 1.3.13

1 point possible (graded)
Consider the following code:

```
g = str(3.14.15)
h = 9/int(0.1234)
j = float(3.4)
```

Which of the above assignments would result in an error?

- ◯ Assignment to `g` only.
- ◯ Assignment to `h` only.
- ◯ Assignment to `j` only.
- ◯ Assignment to exactly two of `g`, `h`, `j`.
- ◯ All three.

Local Variables and Functions

An important note about functions. When you define variables within a function, the **scope of the variable**, that is, where the variable is defined, is limited to the function itself. There is an exception to this with local functions but since we discourage these, the above statement is true for this course.

As an example, consider the following piece of code:

```python
def foo(n):
  a = 12
  return a + n
x = foo(3)
```

When the program is running, the visualizer below creates a box for `foo(3)`. Think of this as a universe that is created with the purpose of computing `foo(3)`. Inside that box, you will notice a variable `n` that is set to 3. Then, once `a = 12` is called, a box labelled `a` is created containing the number 12. Then a return value is created and returned (15 in this case) and the universe for the function disappears from memory as its job is completed. This value 15 is then stored in `x` outside of your code function. Note that you cannot access `a` from outside the function since `a` no longer exists once the function has finished execution. We will see this idea again in future modules. I encourage you however to play around with this code (and create your own examples) in the visualizer element below.

```python
def foo(n):
  a = 12
  return a + n
x = foo(3)
```

Visualize Reset Code

The next video contains 1 question.

If you are having problems viewing the in video questions, please try the following:

- Make sure you are using a course approved browser
- Do not watch the videos at more than 1.0x speed
- Do not watch the videos in full screen.
- Do not scrub the videos (fast forward/rewind to find the videos).
- Try clearing your cache and rewatching the video.

There are not many videos in this course so we do expect you to take the time to actually watch them in their entirety.

Default Parameters

In Python, default parameters are allowed such as:

```python
def not_allowed(x = 4):
  return x + 1
z = not_allowed()
```

However, for this course **we are prohibiting the use of these default parameters**. Not all languages allow this. Futher, later when we introduce lists, default parameters behave in a very unique way which causes misconceptions. Lastly, often times default parameters allow you to miss out on good learning opportunities. For these reasons I mention this here to explicitly state this is not allowed on **any** assignment and will not reference the concept again.

Final Thoughts

In this lesson, we introduced functions. Throughout the course, your goal will frequently be to complete the body of such a function. On assignments, we will also expect you to include the design recipe for all required functions. We will introduce this in the next lesson and finish off by introducing the `math` module which contains many non-standard and useful mathematical functions.