Lab 1: Functions and Control

lab01.zip (lab01.zip)

Due at 11:59pm on Friday, 02/01/2019.

Starter Files

Download lab01.zip (lab01.zip). Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok (ok) autograder.

Submission

By the end of this lab, you should have submitted the lab with `python3 ok --submit`. You may submit more than once before the deadline; only the final submission will be graded. Check that you have successfully submitted your code on okpy.org (https://okpy.org/).

- Questions 1-4 must be completed in order to receive credit for the lab.
- Questions 5-7 are **optional**. *It is recommended that you complete these problems if you finish the required portion early or on your own time.*

Quick Logistics Review

Using Python

Using Python

When running a Python file, you can use options on the command line to inspect your code further. Here are a few that will come in handy. If you want to learn more about other Python command-line options, take a look at the documentation (https://docs.python.org/3.4/using/cmdline.html).

- Using no command-line options will run the code in the file you provide and return you to the command line.

```
python3 lab01.py
```

- **`-i`** : The `-i` option runs your Python script, then opens an interactive session. In an interactive session, you run Python code line by line and get immediate feedback instead of running an entire file all at once. To exit, type `exit()` into the interpreter prompt. You can also use the keyboard shortcut `Ctrl-D` on Linux/Mac machines or `Ctrl-Z Enter` on Windows.
  If you edit the Python file while running it interactively, you will need to exit and restart the interpreter in order for those changes to take effect.

  ```
  python3 -i lab01.py
  ```

- **`-m doctest`** : Runs doctests in a particular file. Doctests are surrounded by triple quotes (`"""`) within functions.
  Each test in the file consists of `>>>` followed by some Python code and the expected output (though the `>>>` are not seen in the output of the doctest command).

  ```
  python3 -m doctest lab01.py
  ```

Using OK

Using OK

In 61A, we use a program called Ok for autograding labs, homeworks, and projects. You should have Ok in the starter files downloaded at the start of this lab. For more information on using Ok commands, learn more here (http://cs61a.org/articles/using-ok.html). To use Ok to run doctests for a specified function, run the following command:

```
python3 ok -q <specified function>
```

By default, only tests that did not pass will show up. You can use the `-v` option to show all tests, including tests you have passed:

```
python3 ok -v
```

Finally, when you have finished all the questions in lab01.py (lab01.py), you must submit the assignment using the `--submit` option:

```
python3 ok --submit
```

Topics

Consult this section if you need a refresher on the material for this lab. It's okay to skip directly to the questions and refer back here should you get stuck.

Division

Division

Let's compare the different division-related operators in Python:

True Division: `/` (decimal division)	**Floor Division: `//` (integer division)**	**Modulo: `%` (remainder)**

`>>> 1 / 5`
`0.2`
 	
`>>> 25 / 4`
`6.25`
 	
`>>> 4 / 2`
`2.0`
 	
`>>> 5 / 0`
`ZeroDivisionError`
	`>>> 1 // 5`
	`0`
	
	`>>> 25 // 4`
	`6`
	
	`>>> 4 // 2`
	`2`
	
	`>>> 5 // 0`
	`ZeroDivisionError`
		`>>> 1 % 5`
		`1`
		
		`>>> 25 % 4`
		`1`
		
		`>>> 4 % 2`
		`0`
		
		`>>> 5 % 0`
		`ZeroDivisionError`

Notice that Python outputs `ZeroDivisionError` for certain cases. We will go over this later in this lab under Error Messages.

One useful technique involving the `%` operator is to check whether a number `x` is divisible by another number `y`:

```
x % y == 0
```

For example, in order to check if `x` is an even number:

```
x % 2 == 0
```

Functions

Functions

If we want to execute a series of statements over and over, we can abstract them away into a function to avoid repeating code.

For example, let's say we want to know the results of multiplying the numbers 1-3 by 3 and then adding 2 to it. Here's one way to do it:

```
>>> 1 * 3 + 2
5
>>> 2 * 3 + 2
8
>>> 3 * 3 + 2
11
```

If we wanted to do this with a larger set of numbers, that'd be a lot of repeated code! Let's write a function to capture this operation given any input number.

```
def foo(x):
    return x * 3 + 2
```

This function, called `foo`, takes in a single **argument** and will **return** the result of multiplying that argument by 3 and adding 2.

Now we can **call** this function whenever we want this operation to be done:

```
>>> foo(1)
5
>>> foo(2)
8
>>> foo(1000)
3002
```

Applying a function to some arguments is done with a **call expression**.

Call expressions

A call expression applies a function, which may or may not accept arguments. The call expression evaluates to the function's return value.

The syntax of a function call:

```
  add   (    2    ,    3   )
   |          |         |
operator  operand  operand
```

Every call expression requires a set of parentheses delimiting its comma-separated operands.

To evaluate a function call:

1. Evaluate the operator, and then the operands (from left to right).
2. Apply the operator to the operands (the values of the operands).

If an operand is a nested call expression, then these two steps are applied to that operand in order to evaluate it.

`return` and `print`

Most functions that you define will contain a `return` statement. The `return` statement will give the result of some computation back to the caller of the function and exit the function. For example, the function `square` below takes in a number `x` and returns its square.

```
def square(x):
    """
    >>> square(4)
    16
    """
    return x * x
```

When Python executes a `return` statement, the function terminates immediately. If Python reaches the end of the function body without executing a `return` statement, it will automatically return `None`.

In contrast, the `print` function is used to display values in the Terminal. This can lead to some confusion between `print` and `return` because calling a function in the Python interpreter will print out the function's return value.

However, unlike a `return` statement, when Python evaluates a `print` expression, the function does *not* terminate immediately.

```
def what_prints():
    print('Hello World!')
    return 'Exiting this function.'
    print('61A is awesome!')

>>> what_prints()
Hello World!
'Exiting this function.'
```

> Notice also that `print` will display text **without the quotes**, but `return` will preserve the quotes.

Control

Control

Boolean Operators

Python supports three boolean operators: `and`, `or`, and `not`:

```
>>> a = 4
>>> a < 2 and a > 0
False
>>> a < 2 or a > 0
True
>>> not (a > 0)
False
```

- `and` evaluates to `True` only if both operands evaluate to `True`. If at least one operand is `False`, then `and` evaluates to `False`.
- `or` evaluates to `True` if at least one operand evaluates to `True`. If both operands are `False`, then `or` evaluates to `False`.
- `not` evaluates to `True` if its operand evaluates to `False`. It evaluates to `False` if its operand evalutes to `True`.

What do you think the following expression evaluates to? Try it out in the Python interpreter.

```
>>> True and not False or not True and False
```

It is difficult to read complex expressions, like the one above, and understand how a program will behave. Using parentheses can make your code easier to understand. Python interprets that expression in the following way:

```
>>> (True and (not False)) or ((not True) and False)
```

This is because boolean operators, like arithmetic operators, have an order of operation:

- `not` has the highest priority
- `and`
- `or` has the lowest priority

It turns out `and` and `or` work on more than just booleans (`True`, `False`). Python values such as `0`, `None`, `''` (the empty string), and `[]` (the empty list) are considered false values. *All* other values are considered true values.

Short Circuiting

What do you think will happen if we type the following into Python?

```
1 / 0
```

Try it out in Python! You should see a `ZeroDivisionError`. But what about this expression?

```
True or 1 / 0
```

It evaluates to `True` because Python's `and` and `or` operators *short-circuit*. That is, they don't necessarily evaluate every operand.

Operator	**Checks if:**	**Evaluates from left to right up to:**	**Example**

AND	All values are true	The first false value	`False and 1 / 0` evaluates to `False`
OR	At least one value is true	The first true value	`True or 1 / 0` evaluates to `True`

Short-circuiting happens when the operator reaches an operand that allows them to make a conclusion about the expression. For example, `and` will short-circuit as soon as it reaches the first false value because it then knows that not all the values are true.

If `and` and `or` do not *short-circuit*, they just return the last value; another way to remember this is that `and` and `or` always return the last thing they evaluate, whether they short circuit or not. Keep in mind that `and` and `or` don't always return booleans when using values other than `True` and `False`.

If Statements

You can review the syntax of `if` statements in Section 1.5.4 (http://composingprograms.com/pages/15-control.html#conditional-statements) of Composing Programs.

Tip: We sometimes see code that looks like this:

```
if x > 3:
    return True
else:
    return False
```

This can be written more concisely as `return x > 3`. If your code looks like the code above, see if you can rewrite it more clearly!

While Loops

You can review the syntax of `while` loops in Section 1.5.5 (http://composingprograms.com/pages/15-control.html#iteration) of Composing Programs.

Error Messages

Error Messages

By now, you've probably seen a couple of error messages. They might look intimidating, but error messages are very helpful for debugging code. The following are some common types of errors:

Error Types	**Descriptions**

SyntaxError	Contained improper syntax (e.g. missing a colon after an `if` statement or forgetting to close parentheses/quotes)
IndentationError	Contained improper indentation (e.g. inconsistent indentation of a function body)
TypeError	Attempted operation on incompatible types (e.g. trying to add a function and a number) or called function with the wrong number of arguments
ZeroDivisionError	Attempted division by zero

Using these descriptions of error messages, you should be able to get a better idea of what went wrong with your code. **If you run into error messages, try to identify the problem before asking for help.** You can often Google unfamiliar error messages to see if others have made similar mistakes to help you debug.

For example:

```
>>> square(3, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: square() takes 1 positional argument but 2 were given
```

Note:

- The last line of an error message tells us the type of the error. In the example above, we have a `TypeError`.
- The error message tells us what we did wrong -- we gave `square` 2 arguments when it can only take in 1 argument. In general, the last line is the most helpful.
- The second to last line of the error message tells us on which line the error occurred. This helps us track down the error. In the example above, `TypeError` occurred at `line 1`.

Required Questions

What Would Python Display (Part 1)?

Q1: WWPD: Control

Use Ok to test your knowledge with the following "What Would Python Display?" questions:

```
python3 ok -q control -u
```

```
>>> def xk(c, d):
...     if c == 4:
...         return 6
...     elif d >= 4:
...         return 6 + 7 + c
...     else:
...         return 25
>>> xk(10, 10)
------

>>> xk(10, 6)
------

>>> xk(4, 6)
------

>>> xk(0, 0)
------
```

```
>>> def how_big(x):
...     if x > 10:
...         print('huge')
...     elif x > 5:
...         return 'big'
...     elif x > 0:
...         print('small')
...     else:
...         print("nothin'")
>>> how_big(7)
------

>>> how_big(12)
------

>>> how_big(1)
------

>>> how_big(-1)
------
```

```
>>> n = 3
>>> while n >= 0:
...     n -= 1
...     print(n)
------
```

> *Hint*: Make sure your `while` loop conditions eventually evaluate to a false value, or they'll never stop! Typing `Ctrl-C` will stop infinite loops in the interpreter.

```
>>> positive = 28
>>> while positive:
...     print("positive?")
...     positive -= 3
------
```

```
>>> positive = -9
>>> negative = -12
>>> while negative:
...     if positive:
...         print(negative)
...     positive += 3
...     negative += 3
------
```

Q2: WWPD: Veritasiness

> Use Ok to test your knowledge with the following "What Would Python Display?" questions:
>
> ```
> python3 ok -q short_circuiting -u
> ```

```
>>> True and 13
------

>>> False or 0
------

>>> not 10
------

>>> not None
------
```

```
>>> True and 1 / 0 and False
------

>>> True or 1 / 0 or False
------

>>> True and 0
------

>>> False or 1
------

>>> 1 and 3 and 6 and 10 and 15
------

>>> 0 or False or 2 or 1 / 0
------
```

```
>>> not 0
------

>>> (1 + 1) and 1
------

>>> 1/0 or True
------

>>> (True or False) and False
------
```

Coding Practice

Q3: Fix the Bug

The following snippet of code doesn't work! Figure out what is wrong and fix the bugs.

```
def both_positive(x, y):
    """Returns True if both x and y are positive.

    >>> both_positive(-1, 1)
    False
    >>> both_positive(1, 1)
    True
    """
    return x and y > 0 # You can replace this line!
```

Use Ok to test your code:

```
python3 ok -q both_positive
```

Q4: Sum Digits

Write a function that takes in a nonnegative integer and sums its digits. (Using floor division and modulo might be helpful here!)

```
def sum_digits(n):
    """Sum all the digits of n.

    >>> sum_digits(10) # 1 + 0 = 1
    1
    >>> sum_digits(4224) # 4 + 2 + 2 + 4 = 12
    12
    >>> sum_digits(1234567890)
    45
    >>> x = sum_digits(123) # make sure that you are using return rather than print
    >>> x
    6
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q sum_digits
```

Optional Questions

What Would Python Display (Part 2)?

Q5: WWPD: What If?

> Use Ok to test your knowledge with the following "What Would Python Display?" questions:
>
> ```
> python3 ok -q what_if -u
> ```
>
> **Hint**: `print` (unlike `return`) does *not* cause the function to exit!

```
>>> def ab(c, d):
...     if c > 5:
...         print(c)
...     elif c > 7:
...         print(d)
...     print('foo')
>>> ab(10, 20)
______
```

```
>>> def bake(cake, make):
...     if cake == 0:
...         cake = cake + 1
...         print(cake)
...     if cake == 1:
...         print(make)
...     else:
...         return cake
...     return make
>>> bake(0, 29)
______

>>> bake(1, "mashed potatoes")
______
```

More Coding Practice

Q6: Falling Factorial

Let's write a function `falling`, which is a "falling" factorial that takes two arguments, `n` and `k`, and returns the product of `k` consecutive numbers, starting from `n` and working downwards.

```
def falling(n, k):
    """Compute the falling factorial of n to depth k.

    >>> falling(6, 3)  # 6 * 5 * 4
    120
    >>> falling(4, 0)
    1
    >>> falling(4, 3)  # 4 * 3 * 2
    24
    >>> falling(4, 1)  # 4
    4
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q falling
```

Q7: Double Eights

Write a function that takes in a number and determines if the digits contain two adjacent 8s.

```
def double_eights(n):
    """Return true if n has two eights in a row.
    >>> double_eights(8)
    False
    >>> double_eights(88)
    True
    >>> double_eights(2882)
    True
    >>> double_eights(880088)
    True
    >>> double_eights(12345)
    False
    >>> double_eights(80808080)
    False
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q double_eights
```

CS 61A (/)

Weekly Schedule (/weekly.html)

Office Hours (/office-hours.html)

Staff (/staff.html)

Resources (/resources.html)

Studying Guide (/articles/studying.html)

Debugging Guide (/articles/debugging.html)

Composition Guide (/articles/composition.html)

Policies (/articles/about.html)

Assignments (/articles/about.html#assignments)

Exams (/articles/about.html#exams)

Grading (/articles/about.html#grading)