Lab 2: Expressions and Control Structures

Due at 11:59pm on 01/28/2015.

Starter Files

Download lab02.zip. Inside the archive, you will find starter files for the questions in this lab, along with a copy of the 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.

- To receive credit for this lab, you must complete Questions 6, 9, and 10 in lab02.py and submit through OK.
- Questions 1, 2, 3, 4, 5, 7, and 8 (What Would Python Print?) are designed to help introduce concepts and test your understanding.
- Questions 11, 12, and 13 are optional **extra practice**. It is recommended that you complete these problems on your own time.

Table of Contents

Using Python

When running a Python file, you can use **flags** 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 flags, take a look at the documentation.

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

```
python3 lab02.py
```

- `-i`: The `-i` option runs your Python script, then opens an interactive session.

```
python3 -i lab02.py
```

- `-m doctest`: Runs doctests in a particular file. Doctests are marked by triple quotes (`"""`) and are usually located within functions.

```
python3 -m doctest lab02.py
```

Using OK

In 61A, we use a program called OK for autograding labs, homeworks, and projects. You should have downloaded ok at the start of this lab. You can use ok to run doctests for a specified function. For example,

```
python3 ok -q factors
```

You can also use the `-i` option: if an error occurs, an interactive interpreter will open, allowing you to enter Python commands to test out your program:

```
python3 ok -q factors -i
```

By default, only errors will show up. You can use the `-v` option to show all tests, including successful ones:

```
python3 ok -v
```

Finally, when you have finished all the quesitons in lab02.py, you can submit the assignment using the `--submit` option:

```
python3 ok --submit
```

Expressions

Question 1: What would Python print?

Think about what the output of each of the following expressions will be. Then type them into Python to verify your answers!

```
>>> 3
_____

>>> 2 + 3
_____

>>> -16 - -16
_____

>>> 3 * 4 + 1
_____

>>> 3 * (4 + 1)
_____

>>> 2 ** 3
_____
```

Toggle Solution

```
>>> x = 4
>>> 3 + x
_____

>>> x + y
_____

>>> x, y = 1, 2
>>> 3 + x
_____

>>> x + y
_____
```

Toggle Solution

```
>>> from operator import mul, add
>>> mul(3, 4)
______

>>> mul(3, add(4, 1))
______

>>> pow(2, 3)
______

>>> pow(pow(2, 3), abs(-2))
______
```

Toggle Solution

Primitive Expressions

A **primitive expression** requires only a single evaluation step: use the literal value directly. For example, numbers, names, and strings are all primitive expressions.

```
>>> 2
2
>>> 'Hello World!'
'Hello World!'
```

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.

Question 2: What Would Python Print?

```
>>> from operator import add
>>> def double(x):
...     return x + x
...
>>> def square(y):
...     return y * y
...
>>> def f(z):
...     add(square(double(z)), 1)
...
>>> f(4)
_____
```

Toggle Solution

```
>>> def welcome():
...     print('welcome to')
...     return 'hello'
...
>>> def cs61a():
...     print('cs61a')
...     return 'world'
...
>>> print(welcome(), cs61a())
_____
```

Toggle Solution

Division

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

- True Division (decimal division) with the `/` operator:

```
>>> 1 / 4
0.25
>>> 4 / 2
2.0
>>> 11 / 3
3.6666666666666665
```

- Floor Division (integer division) with the `//` operator:

```
>>> 1 // 4
0
>>> 4 // 2
2
>>> 11 // 3
3
```

- Modulo (similar to a remainder) with the `%` operator:

```
>>> 1 % 4
1
>>> 4 % 2
0
>>> 11 % 3
2
```

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

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

Pure and Non-Pure Functions

1. Pure functions have no side effects – they only produce a return value. They will always evaluate to the same result, given the same argument value(s).
2. Non-pure functions produce side effects, such as printing to your terminal or returning different outputs on different invocations of the function (non-determinism).

Later in the semester, we will expand on the notion of a pure function versus a non-pure function.

Question 3: What Would Python Print?

```
>>> x = print(9 + 1)
_____

>>> x == 10
_____

>>> print(print(2))
_____
```

Toggle Solution

```
>>> def om(foo):
...     return -foo
...
>>> def nom(foo):
...     print(foo)
...
>>> nom(4)
```

```
______

>>> om(-4)
______

>>> brian = nom(4)
>>> brian + 1
______

>>> michelle = om(-4)
>>> michelle + 1
______
```

Toggle Solution

```
>>> x = 6
>>> def beep(x):
...     print(x)
...
>>> def boop(x):
...     y = x
...     x = 7
...     print(x)
...
>>> y = beep(x)
______

>>> boop(x)
______

>>> y + beep(8)
______
```

Toggle Solution

Boolean operators

Question 4: What Would Python Print?

What would Python print? Try to figure it out before you type it into the interpreter!

```
>>> a, b = 10, 6
>>> a != 0 and b > 5
______

>>> a < b or not a
______

>>> not not a
______

>>> not (not a or not not b)
______
```

Toggle Solution

Boolean order of operations

What do you think the following expression evaluates to?

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

It turns out that Python interprets that expression in the following way:

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

Using parentheses can be helpful to understand how a program will behave.

Boolean operators, like arithmetic operators, have an order of operation:

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

Short-circuit operators

In Python, `and` and `or` are examples of *short-circuiting operators*. Consider the following code:

```
True or 1 / 0
```

Notice that if we just evaluate `1 / 0`, Python will raise an error, stopping evaluation altogether!

However, the original line of code will not cause any errors — in fact, it will evaluate to `True`. This is made possible due to short-circuiting, which works as follows:

- For `and` statements, Python will go left to right until it runs into the first value that is false-y — then it will immediately evaluate to that value. If all of the values are truth-y, it returns the last value.
- For `or` statements, Python will go left to right until it runs into the first value that is truth-y — then it will immediately evaluate to that value. If all of the values are false-y, it returns the last value.

Informally, false-y values are things that are "empty". The false-y values we have learned about so far are `False`, `0`, `None`, and `""` (the empty string).

Question 5: What Would Python Print?

```
>>> 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
______
```

Toggle Solution

Question 6: Fix the Bug

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

Toggle Solution

```
def both_positive(x, y):
    """
    Returns True if both x and y are positive.
    >>> both_positive(-1, 1)
    False
    >>> both_positive(1, 1)
    True
    """
    "*** YOUR CODE HERE ***"
    return x and y > 0
```

You can test your solution by using OK:

```
python3 ok -q both_positive
```

If statements

Question 7: What Would Python Print?

```
>>> a, b = 10, 6
>>> if a == 4:
```

```
...     6
... elif b >= 4:
...     6 + 7 + a
... else:
...     25
...
______
```

Toggle Solution

```
>>> def abs(x):
...     if x >= 0:
...         return x
...     return -x
...
>>> abs(-5)
______

>>> abs(5)
______
```

Toggle Solution

```
>>> def abs(x):
...     if x >= 0:
...         print(x)
...     print(-x)
...
>>> abs(-5)
______

>>> abs(5)
______
```

Toggle Solution

While loops

Question 8: What Would Python Print?

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

>>> n, i = 7, 0
>>> while i < n:
...     i += 2
...     print(i)
...
_____

>>> # typing Ctrl-C will stop infinite loops
>>> n = 4
>>> while True:
...     n -= 1
...     print(n)
...
_____

>>> n = 2
>>> def exp_decay(n):
...     if n % 2 != 0:
...         return
...     while n > 0:
...         print(n)
...         n = n // 2
...
>>> exp_decay(64)
_____

>>> exp_decay(5)
_____
```

Toggle Solution

Question 9: Factor This II

Define a function `factors(n)` which takes in a number, `n`, and prints out all of the numbers that divide `n` evenly. For example, the factors of 20 are 20, 10, 5, 4, 2, 1.

Toggle Solution

```
def factors(n):
    """Prints out all of the numbers that divide `n` evenly.

    >>> factors(20)
    20
    10
    5
    4
    2
    1
    """
    "*** YOUR CODE HERE ***"
```

You can test your solution by using OK:

```
python3 ok -q factors
```

Question 10: Fibonacci

The Fibonacci sequence is a famous sequence in mathematics. The first element in the sequence is 0 and the second element is 1. The nth element is defined as $F_n = F_{n-1} + F_{n-2}$.

Implement the `fib` function, which takes an integer `n` and returns the `n`th Fibonacci number. Use a `while` loop in your solution.

Toggle Solution

```
def fib(n):
    """Returns the nth Fibonacci number.

    >>> fib(0)
```

```
    0
    >>> fib(1)
    1
    >>> fib(2)
    1
    >>> fib(3)
    2
    >>> fib(4)
    3
    >>> fib(5)
    5
    >>> fib(6)
    8
    """
    "*** YOUR CODE HERE ***"
```

You can test your solution by using OK:

```
python3 ok -q fib
```

Error messages

By now, you've probably seen a couple of error messages. Even though they might look intimidating, error messages are actually very helpful in debugging code. The following are some common types of errors (found at the bottom of an error message):

- **SyntaxError**: Indicates that your code contains improper syntax (e.g. missing a colon after an `if` statement).
- **IndentationError**: Indicates that your code contains improper indentation (e.g. inconsistent indentation of a function body)
- **TypeError**: Indicates an attempted operation on incompatible types (e.g. trying to add a function and an int)
- **ZeroDivisionError**: Indicates an 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 unknown error

messages to see what similar mistakes others have made to help you debug your own code.

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
```

Notice that the last line of the error message tells us exactly what we did wrong - we gave `square` 2 arguments when it only takes in 1 argument. In general, the last line is the most helpful.

Here's a link to an extremely helpful Debugging Guide written by Albert Wu. It is highly recommended that you read this in its entirety! Pay particular attention to the section called "Error Types" (the other sections are fairly involved but will be useful in the larger projects).

Extra Questions

> Questions in this section are not required for submission. However, we encourage you to try them out on your own time for extra practice.

Question 11: Disneyland Discounts

Disneyland is having a special where they give discounts for grandparents accompanying their grandchildren. Help Disneyland figure out when the discount should be given. Define a function `gets_discount` that takes two numbers as input (representing the two ages) and returns `True` if one of them is a senior citizen (age 65 or above) and the other is a child (age 12 or below). You should not use `if` in your solution.

Toggle Solution

```
def gets_discount(x, y):
    """ Returns True if this is a combination of a senior citizen
    and a child, False otherwise.

    >>> gets_discount(65, 12)
    True
    >>> gets_discount(9, 70)
    True
    >>> gets_discount(40, 45)
    False
    >>> gets_discount(40, 75)
    False
    >>> gets_discount(65, 13)
    False
    >>> gets_discount(7, 9)
    False
    >>> gets_discount(73, 77)
    False
    >>> gets_discount(70, 31)
    False
    >>> gets_discount(10, 25)
    False
    """
    "*** YOUR CODE HERE ***"
```

You can test your solution by using OK:

```
python3 ok -q gets_discount
```

Question 12: Factor This

Define a function `is_factor` that checks whether its first argument is a factor of its second argument. We will assume that `0` is not a factor of any number but any non-zero number is a factor of `0`. You should not use `if` in your solution.

Toggle Solution

```
def is_factor(x, y):
    """ Returns True if x is a factor of y, False otherwise.
```

```
    >>> is_factor(3, 6)
    True
    >>> is_factor(4, 10)
    False
    >>> is_factor(0, 5)
    False
    >>> is_factor(0, 0)
    False
    """
    "*** YOUR CODE HERE ***"
```

You can test your solution by using OK:

```
python3 ok -q is_factor
```

Question 13: Factorials

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.

Toggle Solution

```
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 ***"
```

You can test your solution by using OK:

```
python3 ok -q falling
```