1. Instructions

In this lab, you have three tasks:

- Think about what python would display section 3.
- Draw environment diagrams for the code in section 4.
- Complete the required problems described in section 5 and submit your code with Ok, as instructed in lab00. The starter code for these problems is provided in `lab01.py`, which is distributed as part of the homework materials.

Submission: As instructed above, you need to submit your work with Ok by `python3 ok --submit`. You may submit more than once before the deadline, and your score of this assignment will be the highest one of all your submissions.

Readings: You might find the following references to the textbook useful:

- Section 1.2
- Section 1.3
- Section 1.4
- Section 1.5

2. Review

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

2.1 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**.

2.1.1 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 inner operand first in order to evaluate the outer operand.

2.1.2 `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.

2.2 Control

2.2.1 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 evaluates 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

Truthy and Falsey Values: 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.

2.2.2 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`.

2.2.3 If Statements

You can review the syntax of `if` statements in Section 1.5.4 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!

2.2.4 While Loops

You can review the syntax of `while` loops in Section 1.5.5 of Composing Programs.

2.3 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`.

2.4 Environment Diagrams

Environment diagrams are one of the best learning tools for understanding `lambda` expressions and higher order functions because you're able to keep track of all the different names, function objects, and arguments to functions. We highly recommend drawing environment diagrams or using Python tutor if you get stuck doing the WWPD problems below. For examples of what environment diagrams should look like, try running some code in Python tutor. Here are the rules:

2.4.1 Assignment Statements

1. Evaluate the expression on the right hand side of the `=` sign.
2. If the name found on the left hand side of the `=` doesn't already exist in the current frame, write it in. If it does, erase the current binding. Bind the *value* obtained in step 1 to this name.

If there is more than one name/expression in the statement, evaluate all the expressions first from left to right before making any bindings.

Try to paste the following code in the Python tutor and understand each execution step. You can also click this link.

```
x = 10
y = x
x = 20
x, y = y + 1, x - 1
```

2.4.2 Def Statements

1. Draw the function object with its intrinsic name, formal parameters, and parent frame. A function's parent frame is the frame in which the function was defined.
2. If the intrinsic name of the function doesn't already exist in the current frame, write it in. If it does, erase the current binding. Bind the newly created function object to this name.

Try to paste the following code in the Python tutor and understand each execution step. You can also click this link.

```
def f(x):
    return x + 1

def g(y):
    return y - 1

def f(z):
    return x * 2
```

2.4.3 Call Expressions

Note: you do not have to go through this process for a built-in Python function like `max` or `print`.

---

1. Evaluate the operator, whose value should be a function.
2. Evaluate the operands left to right.
3. Open a new frame. Label it with the sequential frame number, the intrinsic name of the function, and its parent.
4. Bind the formal parameters of the function to the arguments whose values you found in step 2.
5. Execute the body of the function in the new environment.

Try to paste the following code in the Python tutor and understand each execution step. You can also click this link.

```
def f(a, b, c):
    return a * (b + c)

def g(x):
    return 3 * x

f(1 + 2, g(2), 6)
```

3. What Would Python Display?

In this section, you need to think about what python would display if the code below were input to a python interpreter.

Question 1: Control

Submit your answers with `python3 ok -q q1 -u`.

In What Would Python Display problems, always follow these rules:

For each of the expressions in the table below, write the output displayed by the interactive Python interpreter when the expression is valued. The output may have multiple lines. Each expression has at least one line of output.

- If an error occurs, write `Error`, but include all output displayed before the error.
- If an expression would take forever to evaluate, write `Forever`.

The interactive interpreter displays the value of a successfully evaluated expression, unless it is `None`.

```
>>> 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! Type `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
------
```

Question 2: Veritasiness

Submit your answers with `python3 ok -q q2 -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
------
```

Question 3: What If?

Submit your answers with `python3 ok -q q3 -u`.

```
>>> 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")
------
```

4. Environment Diagram

In this section, you need to draw the environment diagrams of provided code.

You don't have to submit your answers, which means the questions in this section don't count for your final score. However, they are great practice for future assignments, projects, and exams. Attempting these questions is valuable in helping cement your knowledge of course concepts.

Question 4: Bake Cake

Draw the environment diagram of the following code.

```
def square(x):
    return x * x

def bake(cake, make):
    if cake == 0:
        cake = cake + 1
        print(cake)
    if cake == 1:
        print(make)
    else:
        return cake
    return make

cake = square(0)
bake(cake, 29)
```

You can view the correct environment diagram here.

4. Required Problems

In this section, you are required to complete the problems below and submit your code with Ok to get your answer scored.

Remember, you can use Ok to test your code:

```
$ python3 ok
```

and submit your work when you are done:

```
$ python3 ok --submit
```

Problem 1: Fix the Bug (100pts)

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

```
def both_odd(a, b):
    """Returns True if both a and b are odd numbers.

    >>> both_odd(-1, 1)
    True
    >>> both_odd(2, 1)
    False
    """
    return a and b % 2 == 1 # You can replace this line!
```

Test your implementation with `python3 ok -q both_odd`.

Problem 2: Factorial (100pts)

Write a function that takes a positive integer n and returns its factorial.

Factorial of a positive integer n is defined as

$$n! = \prod_{i=1}^{n} i = 1 \times 2 \times 3 \times \cdots \times n.$$

```
def factorial(n):
    """Return the factorial of a positive integer n.

    >>> factorial(3)
    6
    >>> factorial(5)
    120
    """
    "*** YOUR CODE HERE ***"
```

Test your implementation with `python3 ok -q factorial`.

Problem 3: Is Triangle? (100pts)

Write a function that takes three integers (may be nonpositive) and returns `True` if the three integers can form the three sides of a triangle, otherwise returns `False`.

```
def is_triangle(a, b, c):
    """Given three integers (may be nonpositive), judge whether
the three
    integers can form the three sides of a triangle.

    >>> is_triangle(2, 1, 3)
    False
    >>> is_triangle(5, -3, 4)
    False
    >>> is_triangle(2, 2, 2)
    True
    """
    "*** YOUR CODE HERE ***"
```

Test your implementation with `python3 ok -q is_triangle`.

Problem 4: Number of Nine (100pts)

Write a function that takes a positive integer n and returns the number of 9 in each digit of it. (Using floor division and modulo might be helpful here!)

```
def number_of_nine(n):
    """Return the number of 9 in each digit of a positive
integer n.

    >>> number_of_nine(999)
    3
    >>> number_of_nine(9876543)
    1
    """
    "*** YOUR CODE HERE ***"
```

Test your implementation with `python3 ok -q number_of_nine`.

Problem 5: Min Digit (100pts)

Write a function that takes in a non-negative integer and return its min digit. (Using floor division and modulo might be helpful here!)

```
def min_digit(x):
    """Return the min digit of x.

    >>> min_digit(10)
    0
    >>> min_digit(4224)
    2
    >>> min_digit(1234567890)
    0
    >>> # make sure that you are using return rather than print
    >>> a = min_digit(123)
    >>> a
    1
    """
    "*** YOUR CODE HERE ***"
```

Test your implementation with `python3 ok -q min_digit`.

Test your code for lab01 with `python3 ok`, and submit with `python3 ok --submit`.