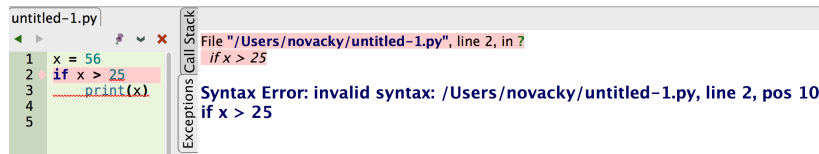


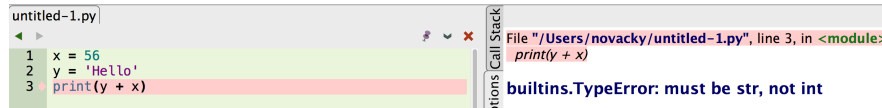
## Lab 6: Program Debugging

There are three kinds of errors that can occur in a program. The first type of error is called a **syntax error**. This kind of error is generated when a programmer does not follow the rules of constructing a correct Python program. Forgetting to type a colon after the question in an `if`-statement, as shown below, is an example of a syntax error.



```
untitled-1.py
1 x = 56
2 if x > 25
3 print(x)
4
5
File "/Users/novacky/untitled-1.py", line 2, in ?
if x > 25
Syntax Error: invalid syntax: /Users/novacky/untitled-1.py, line 2, pos 10
if x > 25
```

The second type of error is called a **runtime error**. This kind of error only makes itself known when the program is running. For example, the program below is constructed properly but the variables `x` and `y` hold values that are incompatible. In line 3, `print(x + y)` says to display the sum of two numbers or display the concatenation of two strings, but `x` and `y` are of incompatible types (they can't be summed nor concatenated together).



```
untitled-1.py
1 x = 56
2 y = 'Hello'
3 print(y + x)
File "/Users/novacky/untitled-1.py", line 3, in <module>
print(y + x)
builtins.TypeError: must be str, not int
```

The third type of error is called a **logical error**. This kind of error is sometimes difficult to detect because the program executes and the program produces results but the results are in error! Python doesn't flag this kind of error nor can Python detect it. It's up to the programmer to ferret out the mistake. These kinds of errors require a technique called **debugging**, the process of eliminating errors from a program. The simplest way to debug a program is to insert `print()` statements in strategic places in a program to reveal the values of important variables. Comparing these values to those that are expected, helps pinpoint the location of an error. In a large program this can be a daunting task. So, the rule of thumb is to enter a section of code, debug it, then and only then move along to the next section of code to be entered. This way you have an idea of where an error is located based on the confidence placed in the debugged code. Commenting out sections of code also helps find where errors are located. Wing allows this to be done by selecting the lines of code to be commented out, then, from the **Source** menu choosing **Toggle Block Comment**. Repeat the process to undo commenting. Another way to debug a program is to use the debugging tools found in **Wing 101**. See the last page of this document for a quick start guide to using Wing's debugging tools. Debugging a computer program is an **art** and a programmer only gets better at it through practice, practice, and more practice!

Use the **Wing** Debugger, your intuition, or place `print()` statements or comment out code to locate where an error occurs in each of these Python programs. Once the error is located, **correct the program** so it runs properly. Submit a zip-file containing each corrected program.

### Problem A.

```
### Program A1
a = input("a: ")
b = input("b: ")

sumsq = 0
for k in range(int(a),int(b)):
    sumsq = sumsq + k*k

print(sumsq)
```

Each of these programs is supposed to sum the squares of the consecutive integers between the inputs **a** and **b** inclusive (note: make sure  $a \leq b$ ). For example, if  $a = 3$  and  $b = 8$ , the sum is  $3^2 + 4^2 + 5^2 + 6^2 + 7^2 + 8^2 = 199$ . Make sure to test your program on more than one or two sets of inputs to be absolutely sure your "fix" is correct in general.

```

### Program A2
a = input("a: ")
b = input("b: ")

sumsq = 0
k = int(a)
while k<=int(b):
    sumsq = sumsq + k*k

print(sumsq)

```

```

### Program A3
a = int(input("a: "))
b = int(input("b: "))

sumsq = 0
for k in range(a,b+1):
    temp = 0
    for u in range(1,k):
        temp = temp + 1

    sumsq = sumsq + temp*temp

print(sumsq)

```

## Problem B.

There are runtime errors in each of the functions below. Can you correct them?

1. This function should return a list of numbers that contains all numbers less than x from num\_list.

```

def collect_those_less_than_x(x, num_list):
    new_list = []
    x = 0
    while x < len(num_list):
        if x < num_list[x]:
            new_list.append(num_list[x])
        x = x + 1
    return new_list

```

2. This function should return a list containing the reverse of the original list.

```

def reverse_list(lst):
    rev_lst = []
    i = len(lst)
    while i > -1:
        rev_lst.append(lst[i])
        i = i + 1
    return rev_lst

```