Lab 1.05

Learning objectives

After completing this lab, students will be able to...

- Demonstrate their understanding of key concepts covered up to this point
- Define and identify: **debugging**, **syntax errors**
- Analyze and respond to error messages

Introduction

In this lab, you will now be explore how to read, analyze, and respond to errors in code. You'll be doing this a lot!! Even seasoned programmers spend most of their time figuring out what was wrong with their first-draft of a program, and repeatedly revising it until it's just right. Python tries to help you out.

A First Error

Read through this code and predict what will be printed out. Remember to go line by line, as if you were the interpreter.

```
favorite_number_str = input("What is your favorite number: ")
birth_month_str = input("What month where you born in: ")

lucky_number = int(favorite_number_str) + int(birth_month_str)
print("Your lucky number is " + str(lucky_number))
```

This code has an error in it. When you run the code, it should generate an error message.

Run the code now by hovering over the brackets to the left of the first line, and pressing the triangle.

Enter `10` as your favorite number and **`March`** as the month you were born in.

You should get something like this:

```
What is your favorite number: 10
What month where you born in: March
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-3-b075f827caf5> in <module>()
      2 birth_month_str = input("What month where you born in: ")
      3
----> 4 lucky_number = int(favorite_number_str) + int(birth_month_str)
      5 print("Your lucky number is " + str(lucky_number))

ValueError: invalid literal for int() with base 10: 'March'
```

Congratulations, you've caused your first error!

Error Reports

When you run a Python program the interpreter executes as much of the program as it can, line by line. If it reaches a line with an error, it halts and reports the problem. The report can be hard to read, but with practice it gets easier.

The most useful line of the report is the last one. In this example, it says there is a `ValueError` involving something with integers, base 10, and the input `March`.

A **ValueError** occurs in Python when a command receives an input that has the right type, but an inappropriate value.

The next most useful part of the report is the partial listing of the code with an arrow pointing to the line where the error occured (line 4 in this example). Something is wrong with the line the is generating `lucky_number`.

Putting these facts together, we can spot the problem: line 4 is trying to make integers out of the two inputs so it can add them together to make `lucky_number`. But the `int()` function doesn't know how to interpret "March" as a base 10 value (it's not smart enough to decide it should be 3, as it's the 3rd month).

Debugging

Debugging is the process of tracking and fixing errors in your code. There's no one recipe for how to do it -- it requires a lot of thinking, reasoning, and testing. But there are many good habits that can help. The first rule of debugging is:

```
%%html
<p style='color:red;font-size:300%;'>Read the error message</p>
```

Let's try applying that rule to another program with a mistake in it. Read the following code and try to predict what it will do. Then run the code by hovering over the brackets to the left of the first line and clicking the triangle.

```
month = "February"
date = 6
year = 2018

print("Today is " + mnth + " " + date + ", " + year + ".")
```

The error report this code generates should look something like this:

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-9-3bbf058bce09> in <module>()
      3 year = 2018
      4
----> 5 print("Today is " + mnth + " " + date + ", " + year + ".")
```

```
NameError: name 'mnth' is not defined
```

Again, the most useful line is the very last one. It tells us there is a **`NameError`**. A `NameError` occurs when Python tries to use the value of a variable but the variable has not yet been introduced into the program. This usually happens because of a typo in the name of the variable, which is the case here.

Read the second part of the error message:

```
name 'mnth' is not defined
```

Where is the arrow pointing? To line 5. And indeed, in line 5 you can see the typo: we've omitted the 'o' from `month`. When we spell it correctly, the program runs without error.

Action at a Distance

Here's a slightly trickier error. Look at this code, see if you can spot the error, then run it.

```
day = "Tuesday"
weater = "rainy"
teacher = "Mr. Miller"

print("Today is " + day + ".")
print("The weather is " + weather)
print("The teacher is " + teacher)
```

The error report should read:

```
NameError                                 Traceback (most recent call last)
<ipython-input-10-92185a5b6c96> in <module>()
      4
      5 print("Today is " + day + ".")
----> 6 print("The weather is " + weather)
      7 print("The teacher is " + teacher)

NameError: name 'weather' is not defined
```

Again, start with the last line. It tells us that there is a `NameError`, and that Python doesn't know about a variable named `weather`. Can you spot the typo?

Next, look at the line indicated by the arrow, line 6:

```
print("The weather is " + weather)
```

Now can you spot the typo?

This case is trickier than before, because line 6 is not the problem. There is no typo in line 6. The problem is back at line 2, where we told Python to create a variable to hold today's weather. That's where we committed a typo:

```
weater = "rainy"
```

Python doesn't know about the spelling of English words. You could name a variable `w` or `lxfb2k` and it would be

Python doesn't know about the spelling of English words. You could name a variable `w` or `ixih2k` and it would be happy. So in this case, it's happy to accept the variable name is the misspelled `weater`. The only problem arises when we spell the variable name differently (as it happens "correctly") in line 6. That's the first time Python tries to get the value of a variable with an unfamiliar name, so that's where it reports a problem.

Common Error Types

There are dozens of error types in Python 3, but in your early days you will encounter just a few:

- `SyntaxError`
- `TypeError`
- `ValueError`
- `NameError`
- `IndentationError`

SyntaxError

A **`SyntaxError`** happens when you make an error in the syntax of your program. Python has no way of making sense of what you wrote, because it is very strict about grammar and spelling. Usually `SyntaxErrors` can be traced back to missing punctuation characters, such as parentheses, quotation marks, or commas

Here are 3 tiny programs each with a `SyntaxError`. Run each one, read the error message, then fix the problem and run it again.

```
# Whoops! Left out the final quotation mark.
drink = "coffee
```

```
# Whoops! Have an extra parenthesis in there.
print("Why can't it understand what I mean..."))
```

```
# Whoops! the second plus sign isn't followed by another string.
# Nothing to print after "grail", so that's a SyntaxError
print("What is your quest? " + "I seek the grail" +)
```

TypeError

A **`TypeError`** occurs when an action is applied to an object of inappropriate type. Some types we've seen are *string*, *integer*, and *float*. Later we'll meet other types like *list*, *set*, and *dictionary*. Some actions make sense for one type but not another. You can add two integers, but you can't add an integer to a string. You can append an item to end of a list, but you can't append an item to the end of a number. Asking Python to do something like that will usually yield a `TypeError`.

For the examples below,

1. read each program
2. identify the error
3. run it
4. read the error message

```
three_string = "three"
three_integer = 3
errorful_six = three_string + three_integer


# Which is bigger, 7 or 4 ?
7 > "four"


# This is the way you define a list in Python
todo_list = ["homework", "exercise", "shower"]

# This is how you do something with each element of a list, one at a time.
for task in todo_list:
  print(task)


# This is not a list
days_in_february = 28

# This doesn't work...
for day in days_in_february:
  print(day)
```

NameError

We've already seen `NameError` in our first debugging examples of this lab. The most common cause is a typo in your code, either in the name of a variable, or in the name of a python command like `print()` or `input()`. Here are 3 examples for you to run:

```
city = "Brookline"
state = "Massachusetts"
qrint("I attend school in " + city + "," + state)


flavor = input("What is your favorite ice cream?")
topping = imput("What is your favorite topping?")
print(flavor + " with " + topping + ". What a great dessert!")


first = "... and the last shall be first."
last = "The first shall be last..., "
temp = first
first = last
last = tmp
print(first + last)
```

ValueError

A **`ValueError`** occurs in Python when a command receives an input that has the right type, but an inappropriate value. For example, if we try to use the `int()` command to convert a string to an integer, but the string is not composed of digits.

```
number_four_hundred = 400
numeral_four_hundred = "400"
words_four_hundred = "four hundred"

# This line is fine, since the string "400" comprises only digits
converted from numeral = int(numeral four hundred)
```

```
# This line is not fine, since the string "four hundred" has non-digits in it.
converted_from_words = int(words_four_hundred)
```

IndentationError

Python is surprisingly picky about blank spaces at the start of a line of code. An **`IndentationError`** occurs if your program violates one of Python's rules about how many spaces or tabs occur at the beginning of a line.

Until now, we haven't had any reason to start a line with a blank space. But watch what happens if we insert one just as an experiment:

```
print("beginning")
 print("middle")
print("end")
```

The leading space on the second line gives Python fits. It displays the error:

```
      File "<ipython-input-15-b38070a7942d>", line 2
        print("middle")
        ^
    IndentationError: unexpected indent
```

To fix it, simply remove the space and all will be well again.

The reason Python is so picky about leading whitespace is that is uses indentation to indicate where blocks of code begin and end. For example, these two programs do differnet things (run them to verify this)

```
# Version 1: final line indented 2 spaces
n = 0
while n < 4:
  n = n + 1
  print("n is: " + str(n))
  print("Pay attention to indentation!")
```

```
# Version 2: final line *not* indented at all
n = 0
while n < 4:
  n = n + 1
  print("n is: " + str(n))
print("Pay attention to indentation!")
```

In the first version, the final line is indented 2 spaces. That tells Python to consider it part of the instructions to exectue so long as `n` is less than 4. So in version 1, the warning to attend to indentation is printed 4 times.

In the second version, the final line is **not** indented at all. That tells python **not** to consider it part of the instructions to execute repeatedly. So in version 2, the warning is printed just once, after all the repeats of printing `n`.

In most other computer languages, programmers use the curly brace characters `{` and `}` to show where blocks begin and end:

```
# This is not Python. Just showing where curly braces would go if it
```

```
# this is not Python. Just showing where curly braces would go if it
followed
# the style of languages that don't care about white space.

# Version 1: final line inside curly braces
n = 0
while n < 4 {
  n = n + 1
  print("n is: " + str(n))
  print("Pay attention to indentation!")
}

# Version 2: final line outside curly braces
n = 0
while n < 4 {
  n = n + 1
  print("n is: " + str(n))
}
print("Pay attention to indentation!")
```

Python uses whitespace as an indication of block beginnings and endings, and looks for a 2 space indent. But what happens if you give it a 1 space indentation at an unexpected time? You get an error:

```
# Version 3: final line intented just 1 space -- that's an error
n = 0
while n < 4:
  n = n + 1
  print("n is: " + str(n))
 print("Pay attention to indentation")
```

We'll talk about this more when we learn what blocks are actually used for. In the meantime make sure all your code is lined up with the code above it that you want to be run in sequence.

Here's two more indentation errors for you to inspect.

```
a = "silly"
b = "hogwash"
 temp = a
a = b
b = temp
```

```
  print("first line indent")
  print("is fine since nothing is above")
 print("outdent is a problem")
```

```
a = 1
 b = 2
 c = 3
 d = 4
 print(c)
```

Find and Fix Errors

Find and Fix Exercises

Now it's time for you to find bugs in some larger programs. These programs may use techniques and parts of Python you haven't encountered yet. Don't worry about that. Run the program, read the error message, look at the type of the error, look at the line it points you to, fix the problem.

Each program should run smoothly when you fix the problem.

Count by Fives

```
for i in range(5, 105, 5):
    printt(i)
```

Adding the First N Numbers

```
prompt = """
I can add up all the numbers from 1 to any number you like.
Enter a number -->
"""
n_str = input(prompt)
n = int n_str)

# Start the total at 1 and the range at 2 so our output looks nice
# without boring 0+0=0, 0+1=1
# End the range at n+1 so we include n in the total
total = 1
for number in range(2, n+1):
    print("%d+%d=%d" % (total, number, number + total))
    total += number
print("The answer is %d" % total)
```

Typing Test

```
import random

targets = ["apple", "banana", orange, "pear"]
for target in targets:
    entry = None
    while entry != target:
        entry = input("Type \"%s\":\n----> " % target)
    print("You got it! Now try another.\n")
```

Word Scramble

When you run this program, the error message will offer you a suggestion of how to fix it. In this case, that suggestion is not on target -- try to fix the error in a more natural, straightforward way. Hint: you are missing just two characters to make it right.

```
import random

print("Welcome to Word Scramble!\n\n")
print("Try unscrambling these letters to make an english word.\n")
```

```
words = ["apple", "banana", "peach", "apricot"]

for i in range(1,3):
  word = random.choice(words)
  letters = list(word)
  random.shuffle(letters)
  scramble = ''.join(letters)

  print("Scrambled: %s" % scramble)
  guess = input("What word is this? ")
  if guess == word:
    print "\nThat's right!\n"
  else:
    print("\nNo, the word was %s\n" % word)
```

Scrabble Tiles

This one is tricky -- you might want to reread **Action at a Distance** above. Then check out how the variable **`total`** is given its initial value. Does that look right to you?

```
value_of_tile = {'A': 1, 'B': 3, 'C': 3, 'D': 2, 'E': 1, 'F': 4, 'G': 2,
                 'H': 4, 'I': 1, 'J': 8, 'K': 5, 'L': 1, 'M': 3, 'N': 1,
                 'O': 1, 'P': 3, 'Q': 10, 'R': 1, 'S': 1, 'T': 1, 'U': 1,
                 'V': 4, 'W': 4, 'X': 8, 'Y': 4, 'Z': 10}

print("Enter a word and I'll compute the value in Scrabble.")

word = input("Enter a word: ")
total = "zero"
print()
for letter in word.upper():
    value = value_of_tile[letter]
    print("%s is worth %d" % (letter, value))
    total = value + total
print()
print("The total is %d" % total)
```

Rock, Paper, Scissors

It's common for your code to have more than one error.

In this program, there are 3 separate bugs. Python will stop running your program as soon as it reaches the first error. When you fix that and re-run, the error report will change and you'll reach the second error, and so on until you find and fix all three.

Read the error reports so you know whether your change fixed the error!!

```
import random

choices = ["rock", "paper", "scissors"]

user_choice = input("rock, paper, or scissors? ")
if (user_choice not in choices):
    print("Sorry, you must type \"rock\", \"paper\", or \"scissors\" (all lower-case).")
    print("You typed: \"" + user_choice + "\".")
    raise ValueError("That's not 'rock', 'paper', or 'scissors'")

computer_choice = random.choice(choices)

print("\n")
print("I chose " + computer choice + ".")
```

```
print("You chose " + user_choice + ".")
print("\n")

if (user_choice == "rock" and computer_choce == "paper"):
    print("I win!")
elif (user_choice == "rock" and computer_choice == "scissors"):
    print("You win!")
elif (user_choice == "paper" and computer_choice == "scissors"):
    print("I win!")
elif (user_choice == "paper" and computer_choce == "rock"):
    print("You win!")
elif (user_choice == "scissors" and computer_choce == "rock"):
    print("I win!")
elif (user_choice == "scissors" and computer_choice == "paper"):
    print("You win!")
else:
    print("It's a tie!")

print('Let's play again")
```

Go Bug Someone Else!

Easy as Pie

Now it's time to challenge your neighbor!

First run this program to confirm that it works. Then introduce a bug by adding or removing just a few characters. Do it in secret, and then show your neighbor the error message and ask them to find the bug you inserted.

Try to make it one of the error types we've discussed above, or just try deleting a single random character and see if you get an error message.

Take turns creating bugs for each other to find. If you can't figure out how to get the program working again, just reload this notebook and it should restore the original version.

If you get bored with this program, try inserting your own errors into one of the **Find and Fix** examples above for your neighbor to find.

Good luck stumping your neighbor!!

```
# This computes the Leibniz formula for pi
# pi/4 = 1 - 1/3 + 1/5 - 1/7 + 1/9 ...
# It is a very slowly converging series.
# 10000 terms gets us 3.141(wrong digits after)

one_fourth_pi = 0.0
for term in range(10000):
    denominator = 2*term + 1
    fraction = 1.0/denominator
    if term % 2 == 0:
        one_fourth_pi += fraction
    else:
        one_fourth_pi -= fraction
print("Pi is approximately: " + str(4*one_fourth_pi))
```