Activity 1.3.4 Nested Branching and Input

Introduction

Most useful programs have a way to get input from the user, make a decision, and do different things depending on the input. Programs usually have a way to communicate output back to the user.

Think of a program or app you've used. What was the input? What was the output? Did the program's behavior depend on the input?

Procedure

1. Form pairs as directed by your teacher. Meet or greet each other to practice professional skills. Launch Canopy and open an editor window.

Part I. Nested `if` structures and testing

2. The `if`–`else` structures can be **nested**. The indentation tells the *Python*® interpreter what blocks of code should be skipped under what conditions. Paste the code below into your *Python* file. The line numbers will be different.

```
 1  def food_id(food):
 2      ''' Returns categorization of food
 3
 4      food is a string
 5      returns a string of categories
 6      '''
 7      # The data
 8      fruits = ['apple', 'banana', 'orange']
 9      citrus = ['orange']
10      starchy = ['banana', 'potato']
11
12      # Check the category and report
13      if food in fruits:
14          if food in citrus:
15              return 'Citrus, Fruit'
16          else:
17              return 'NOT Citrus, Fruit'
18      else:
19          if food in starchy:
20              return 'Starchy, NOT Fruit'
21          else:
22              return 'NOT Starchy, NOT Fruit'
```

```
In []: food_id('apple')
'NOT Citrus, Fruit'
```

a. Did this return value result from line 15, 17, 20, or 22 (refer to line numbers shown above)?

b. Every input will cause only one of the following lines of code to be executed.
   i. What input will cause line 15 to be executed?

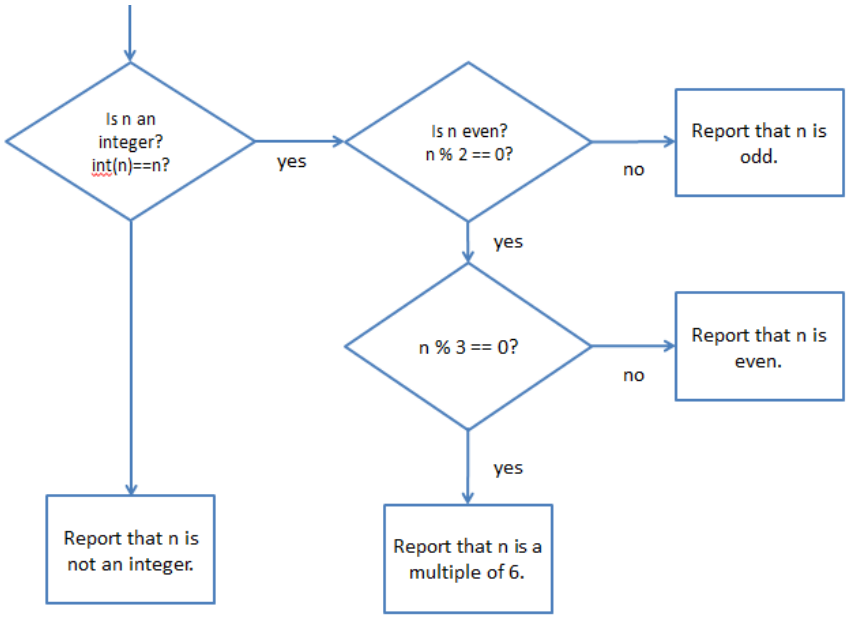
   ii. What input will cause line 17 to be executed?

   iii. What input will cause line 20 to be executed?

   iv. What input will cause line 22 to be executed?

c. Bananas are starchy, and the program "knows" it. Explain why line 20 will never result in bananas being reported as starchy.

3. Define a function `f(x)` that implements the **flow chart below**. A flow chart is another way to represent an algorithm; input and output are in rectangles, and branching decisions are in diamonds. The exercise illustrates the `%` operator, which identifies the remainder after division. As an example, `13 % 4` is 1, since 13÷ 4 is 3 remainder 1.




```
In []: f(12)
'The number is a multiple of 6.'
```

4. Write a set of test cases *(hint: x-values)* which would visit all the code.

Part II: The `raw_input()` function, type casting, and print() from *Python* 3

5. To get input from the user of a program, we normally use a **graphical user interface** (GUI). That is the subject of Lesson 1.3. Beginners often want a simple way to obtain text input. *Python* uses the `raw_input(prompt)` command. It has some annoying behavior that we have to deal with for now — it

always returns a string even when numeric type is appropriate. In addition iPython ignores Ctrl-C interrupts with raw_input(), so infinite loops will require restarting the *Python* kernel. Finally the prompt doesn't appear until the user starts typing. That said, here's how you use it:

```
In []: a = raw_input('Give me a number between 5 and 6: ')
Give me a number between 5 and 6: 5.5
```

Even though the user typed a number, `raw_input()` returned a string. You can see that as follows.

```
In []: a
Out[]: u'5.5'

In []: type(a)
Out[]: unicode
```

The variable `a` has a variable type that is a string. Keyboard input might be encoded as a `unicode` type, as shown above, or as a `str` type, but either way, it is a string of characters. (**Unicode** is a set of characters that includes all of the world's written languages. It is encoded with UTF-8, an extension of ASCII. The `u` in `u'5'` indicates that the string returned by the `raw_input()` command is a Unicode string.)

To use numeric values from the input, you have to turn the string into an `int` or a `float`. This will **raise an error** if the user didn't provide an `int` or a `float`. There are commands – not covered in this course – that **catch** the error so that it doesn't continue up to the *Python* interpreter and halt the program. For now, however, we can live with an error if the user does something unexpected.

To convert from a string to a number, you can use the `int()` function or the `float()` function. Forcing a value to be converted to a particular type is called **type casting**. Continuing from `a` being `'5.5'` above,

```
In []: int(a)
ValueError: invalid literal for int() with base 10: '5.5'

In []: float(a)
Out[]: 5.5

In []: int(float(a))
Out[]: 5
```

You can also type cast a number into a string:

```
In []: b = 6

In []: a + b
TypeError: cannot concatenate 'str' and 'int' objects

In []: a + str(b)
Out[]: '5.56'

In []: float(a) + b
```

```
Out[]: 11.5
```

a) Explain the difference between + as concatenation and + as numeric addition.

6. The following code picks a random number between 1 and 4 (**inclusive,** meaning it includes both 1 and 4) and lets the user guess once. In part b below, you will modify the program so that it indicates whether the user guessed too low, too high, or correctly.

```
 1  from __future__ import print_function # must be first in file
 2  import random
 3
 4  def guess_once():
 5      secret = random.randint(1, 4)
 6      print('I have a number between 1 and 4.')
 7      guess = int(raw_input('Guess: '))
 8      if guess != secret:
 9          print('Wrong, my number is ', secret, '.', sep='')
10      else:
11          print('Right, my number is', guess, end='!\n')
```

```
In []: guess_once()
I have a number between 1 and 4 inclusive.
Guess: 3
Right, my number is 3!

In []: guess_once()
I have a number between 1 and 4 inclusive.
Guess: 3
Wrong, my number is 4.
```

In line 9, `print('Wrong, my number is ', secret, '.', sep='')` has four arguments: three strings and a keyword=value pair. This is `print(s1, s2, s3, sep='')`. If the `sep=''` were not there, this would print the three strings separated by spaces. The separator is a space—by default—for the output of `print()`, but the function offers a keyword for setting the separator to a different value. The argument `sep=''` sets it to **the null string,** i.e., the empty string.

You'll learn about the random module in the next activity.

```
 2  import random
 5  secret = random.randint(1, 4) # randint() picks including endpts
```

a. Explain how line 11 works, using the explanation of line 9 as a model.

b. Modify the program to provide output as shown below.

```
In []: guess_once()
I have a number between 1 and 4 inclusive.
Guess: 3
Too low - my number was 4!
```

```
In []: guess_once()
I have a number between 1 and 4 inclusive.
Guess: 3
Too high, my number was 2!

In []: guess_once()
I have a number between 1 and 4 inclusive.
Guess: 1
Right on! I was number 1!
```

7. Create a function `quiz decimal(low, high)` that asks the user for a number between `low` and `high` and tells them whether they succeeded.

```
In []: quiz_decimal(4, 4.1)
Type a number between 4 and 4.1:
4.5
No, 4.5 is greater than 4.1

In []: quiz_decimal(4, 4.1)
Type a number between 4 and 4.1:
4.05
Good! 4 < 4.05 < 4.1
```

Conclusion

1. What is the relationship between if-structures and glass box testing?

2. Nested if-else structures can contain many blocks of code. How many of those blocks of code might be executed?

3. What does a test suite do, and why do you think programmers often write test suites first, before they've even written the functions that will be tested?