Lab 2 Summary

Working with Text and Making Choices

In this minilab, we will use R to create some simple datasets and start to apply some simple aspects of exploratory data analysis.

- Study *Think Python* Chapter 8: Strings
- Study *Think Python* Chapter 5: Conditionals and Recursion
- *Suggested reading: Think Python* Section 3.3: Math functions

1. Creating strings of characters

Computers were invented to do arithmetic.

But these days, computers spend a lot of time processing text.

In Python, text is represented as a *string*, which is a sequence of *characters* (letters, digits, and symbols).

The type whose values of sequences of characters is *str*.

In Python, we indicate that a value is a string by putting either single or double quotes around it. The opening and closing quotes must match.

```
>>> 'Isaac Newton'
'Isaac Newton'
>>> "Albert Einstein"
'Albert Einstein'
```

Strings can contain any number of characters, limited only by computer memory.

The shortest string is the *empty string*, containing no characters at all.

```
>>> ""
''
```

Operations on Strings

```
>>> len('Isaac Newton')
12
>>> len(' ')
1
>>> len('')
0
>>> 'Isaac' + 'Newton'
'IsaacNewton'
```

When + operates on two strings it is the *concatenation character*.

We can convert strings to numbers and vice versa.

```
>>> int('123')
123
>>> string(123)
'123'
>>> float('3.1415')
3.1415
```

When * operates on a string an integer, we repeat the string.

```
>>> 'ABC' * 4
'ABCABCABCABC'
```

Special characters in Strings

So single and double quotes are equivalent EXCEPT for strings that contain quotes.

```
>>> 'this won't work'
```

```
SyntaxError: invalid syntax
>>> "that's better"
"that's better"
>>> 'She said, "That is better."'
'She said, "That is better."'
```

Another way is to use an *escape sequence*, to escape from Python's usual syntax rules for a moment.

When Python sees a backslash inside a string, it means that the next character represents something that Python normally uses for another purpose, such as marking the end of a string.

\'	single quote	\t	tab
\"	double quote	\n	newline
\\\\	backslash		

Multiline strings (triple quotes)

```
>>> '''Line 1
Line 2
Line 3'''
'Line 1\nLine 2\nLine 3'
```

Printing information

Use the built-in function *print* to print values to the screen.

```
>>> print(1 + 2)
3
>>> print('The quick brown fox')
The quick brown fox
>>> radius = 3
>>> print('Diameter of the circle is', 2 * radius)
Diameter of the circle is 6
```

Getting information from the keyboard

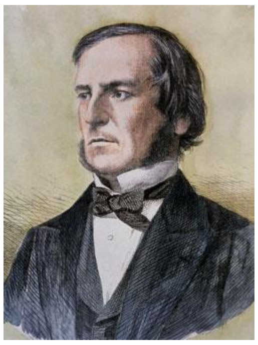
```
>>> country = input()
United Kingdom
>>> country
'United Kingdom'
>>> type(country)
<class 'str'>
>>> population = input('Enter the population: ')
Enter the population: 66859058
>>> population
'66859058'
>>> type(population)
<class 'str'>
>>> population = int(input())
```

▶ We will look at operations on strings later when we look at methods.

2. A Boolean type

In Python, there is a type called *bool* which has only two values: **True** and **False**.

George Boole (1815 – 1864)



Boolean operators

There are only three basic Boolean operators: **and**, **or**, and **not**.

```
>>> True and True
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
```

This information can be summarised in a *truth table* for the logical operation AND.

A	**B**	**A and B**
True	True	True
True	False	False
False	True	False
False	False	False

We can build similar truth tables for OR and NOT.

```
>>> True or True
True
>>> True or False
```

```
True
>>> False or True
True
>>> False or False
False
>>> not True
False
>>> not False
True
```

Precedence

not	highest precedence
and	
or	lowest precedence

3. Relational operators

True and False are values.

These values are usually created in expressions.

Each of the relational operators (<, >, <=, >=, ==, !=) produces a True or False.

>	greater than	>=	greater than or equal to
<	less than	<=	less than or equal to
==	equal to	!=	not equal to

```
>>> 45 > 34
True
>>> 45 < 34
False
>>> 45 != 34
True
>>> 45 == 34
```

```
False
```

Or how about ...

```
>>> def is_positive(x):
      return x>0

>>> is_positive(3)
True
>>> is_positive(0)
False
>>> is_positive(-10)
False
```

Combining Comparisons

```
>>> x = 2
>>> y = 5
>>> z = 8
>>> (x < y) and (y < z)
True
```

Short-Circuit Evaluation

```
>>> 1/0
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    1/0
ZeroDivisionError: division by zero
>>> (2 < 3) or (1/0 < 6)
True
```

Comparing Strings

```
>>> 'A' < 'a'
True
>>> ord('A')
65
>>> ord('a')
97
>>> 'abc' < 'abd'
True
>>> 'Sep' in '23 Sep 2020'
True
```

The order of the characters is given by Unicode.

4. Choosing which statements to execute

Remember that you can enter your Python code into an editor, save it with a .py extension and then run it from the commandline by giving the filename after the word python.

An if statement lets you change how your program behaves based on a condition.

The condition is an expression that evaluates to a Boolean value.

The block of statements inside an if must be indented. The standard indentation for Python is four spaces.

If the condition evaluates to True, the statements in the block are executed; otherwise they are not.

```
ph = float(input('Enter the pH level of the chemical: '))
if (ph < 7.0):
    print(ph, ' is acidic')
if (ph > 7.0):
    print(ph, ' is basic')
```

Here both conditions are evaluated, even though we know that only one of the blocks can be executed.

We can merge both cases by using the elif keyword (which standard for “else if”).

```
ph = float(input('Enter the pH level of the chemical: '))
if (ph < 7.0):
    print(ph, ' is acidic')
elif (ph > 7.0):
    print(ph, ' is basic')
```

Nested if statements

An if statement’s block can contain any type of Python statement, which implies that it can include other if statements.

An if statement inside another is called a *nested* if statement.

```
a = 3
b = 5
if (a < 4):
    if (b < 4):
        print('both small')
    else:
        print('a small, b large)
else:
    if (b < 4):
        print('a large, b small)
    else:
        print('both large')
```

5. Modules

A *module* is a collection of variables and functions that are grouped together in a single file.

The variables and functions in a module are usually related to each other in some way.

To gain access to the variables and function from a module, you have to *import* it.

The math module provides access to lots of mathematical functions and constants (such π).

```
>>> import math
>>> type(math)
<class 'module'>
```

We can now use all the standard mathematical functions.

```
>>> sqrt(9)
Traceback (most recent call last):
  File "<pyshell#56>", line 1, in <module>
    sqrt(9)
NameError: name 'sqrt' is not defined
```

We must tell Python explicitly to look for the function in the module math by combining the module's name with the function's name using a dot.

```
>>> math.sqrt(9)
3.0
```

Python finds math in the current namespace, looks up the module object that math refers to, finds function sqrt inside the module, and then executes the function call following the standard rules.

Modules contain more than just functions.

```
>>> math.pi
3.141592653589793
>>> radius = 4
>>> area = math.pi * radius**2
>>> area
50.26548245743669
```

In order to save code for later use, you can put it in a file with a .py extension and run it.

You can put several functions in one module.

Python executes modules as it imports them. So if you some statements that are not in functions in your module then they will be executed.

There is a nice feature of Python that we can use to tell whether a module is imported or run.

```
if (__name__ == '__main__'):
    # some statements that execute if run but not if imported
```

6. Methods and Classes

We have seen a lot of functions: built-in functions, functions inside modules, and functions that we have defined ourselves.

A *method* is another kind of function that is attached to a particular type.

There are str methods, int methods, bool methods, and more. Every type has its own set of methds.

We can use str to call a method in class str, much like we call a function in module math.

The main difference is that ALL methods in class str require a string as the first argument.

```
>>> str.capitalize('mary had a little lamb')
'Mary had a little lamb'
>>> str.upper ('mary had a little lamb')
'MARY HAD A LITTLE LAMB'
>>> str.center('middle',10)
'  middle  '
>>> str.count('I scream, you scream, we all scream for ice cream','scream')
3
```

```
>>> str.split('I scream, you scream, we all scream for ice cream')
['I', 'scream,', 'you', 'scream,', 'we', 'all', 'scream', 'for', 'ice', 'cream']
```

Notice the American spelling of capitalize and center.

The last output from split is a *list* of words.

Calling methods the object-oriented way

Every method in class str requires a string as the first argument.

More generally, every methods in any class requires an object of that class as the first argument.

Therefore, Python provides a shorthand form for calling a method where the object appears first and then the method call.

```
>>> 'Mary had a little lamb'.capitalize()
'Mary had a little lamb'
>>> 'middle'.center(10)
'  middle  '
>>> 'I scream, you scream, we all scream for ice cream'.count('scream')
3
```

You can find a description of many string methods at: https://www.w3schools.com/python/python_ref_string.asp

7. Random numbers

Random numbers (actually pseudo-random numbers) are provided by the Python module (library) *random*.

```
import random
# Roll a dice
print(random.randint(1,6))
```

If you repeatedly call randint then you will get a “fresh” random value each time.

```
import random


# Roll a dice 12 times
print(random.randint(1,6))
print(random.randint(1,6))
print(random.randint(1,6))
print(random.randint(1,6))
print(random.randint(1,6))
print(random.randint(1,6))
print(random.randint(1,6))
print(random.randint(1,6))
print(random.randint(1,6))
print(random.randint(1,6))
print(random.randint(1,6))
print(random.randint(1,6))
```

Summary

Strings

- Python uses type str to represent text as sequences of characters.
- Strings are created by placing pairs of single quotes (') or double quotes (") around the text.
- Use triple quotes (' ' 'somthing ' ' ') to create a string that crosses multiple lines.
- Special characters are represented using escape sequences that being with a backslash.
- Values can be printed using built-in function print().
- Input can be provided by the user using built-in function input().

Making choices

- Python uses Boolean values (True and False) to represent what is true and what isn't. Programs can combine these values using three operators: not, and, and or.
- Relational operators such as "equals" and "less than" compare values and produce a Boolean result.
- When different operators are combined in an expression, the order of precedence from highest to lowest is arithmetic, relational, and then Boolean.
- if statements control the flow of execution. As with function definitions, the bodies of if statements are indented, as are the bodies of elif and else clauses.

Modules

- A module is a collection of functions and variables grouped together in a file. To use a module, you must firt import it using import. After it has been imported, you refer to its contents using dot notation.
- Variable __name__ is created by Python and can be used to specify that some code should only run when the module is run directly and not when the module is imported.

Methods

- Classes are like modules, except that classes contain methods and modules contain functions.
- Methods are like functions, except that the first argument must be an object of the class in which the method is defined.
- Method calls in the form 'may the force be with you'.capitalize()
  are shorthand for str.capitalize('may the force be with you')
- Methods beginning and ending with two underscores are considered special by Python, and they are triggered by particular syntax.