Practical 2 - Strings, Files and Exceptions

Did you finish last week's work? If not, make sure to complete it during the week. If you do not understand anything, bring those questions to your tutor at the start of the following week.

Walkthrough Example

String Formatting

The **str.format()** method lets us format strings using placeholders and format specifiers in a way that's very powerful and will make a lot of sense once you get used to it.

Sometimes the best way to start learning this sort of thing is to see some useful examples, so copy the following examples (all in one go) into a new Python file and run the code:

```python
name = "Gibson L-5 CES"
year = 1922
cost = 16035.40

# The 'old' manual way to format text:

print("My guitar: " + name + ", first made in " + str(year))

# A better way using str.format():

print("My guitar: {}, first made in {}".format(name, year))
print("My guitar: {0}, first made in {1}".format(name, year))
print("My {0} was first made in {1} (that's right, {1}!)".format(name, year))

# Formatting currency:

print("My {} would cost ${:,.2f}".format(name, cost))

# Aligning columns:

numbers = [1, 19, 123, 456, -25]
for i in range(len(numbers)):
    print("Number {0} is {1:>5}".format(i + 1, numbers[i]))
```

Nice! Notice:
- You can leave out the positional arguments, that is the numbers inside the {}, if you just want to use the default order.
- You can also repeat values by repeating the positional arguments.
- And you can do lots of formatting by using the string formatting 'mini language'; details come after the **:**

Want to read more about it? https://docs.python.org/3/library/string.html#formatstrings

Random Numbers

We are going to write some programs using random numbers… but how do we generate random numbers? Python has a number of random functions - contained within the **random** module. Unlike the built-in functions (**print()**, **input()**, etc) the random functions are *not* built-in but need to be **imported**. Modules are reusable collections of functions, classes and variables (constants) related to a specific topic (e.g. maths, operating system services, handling dates and times). Python has a useful built-in function for finding out about the local scope of something, called **dir()**.

Launch a Python **console** (in PyCharm, simply click in the footer/status bar where it says "Python Console") and try the following:

```
>>> dir(str)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__',
'__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold',
'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

Look carefully! You can see all of those useful **str** methods such as **upper()**, **startswith()**, **isdecimal()**.

Next try running the **dir()** function with **random** as the argument:

```
>>> dir(random)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'random' is not defined
```

This doesn't work since random is a module that needs to be **imported**. Now try it like this:

```
>>> import random
>>> dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST', 'SystemRandom',
'TWOPI', '_BuiltinMethodType', '_MethodType', '_Sequence', '_Set', '__all__',
'__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', '_acos', '_ceil', '_cos', '_e', '_exp', '_inst', '_log',
'_pi', '_random', '_sha512', '_sin', '_sqrt', '_test', '_test_generator', '_urandom',
'_warn', 'betavariate', 'choice', 'expovariate', 'gammavariate', 'gauss', 'getrandbits',
'getstate', 'lognormvariate', 'normalvariate', 'paretovariate', 'randint', 'random',
'randrange', 'sample', 'seed', 'setstate', 'shuffle', 'triangular', 'uniform',
'vonmisesvariate', 'weibullvariate']
```

This shows you all of the names in the module - most of which are functions you can use. To use functions from a module that's been imported like this you need to *qualify* the name - e.g. use **random.randint** not just **randint**. Use **help()** to find out about a couple of these functions:

```
>>> help(random.randint)
Help on method randint in module random:

randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.

>>> help(random.randrange)
Help on method randrange in module random:
```

```
randrange(start, stop=None, step=1, _int=<class 'int'>) method of random.Random instance
    Choose a random item from range(start, stop[, step]).
    This fixes the problem with randint() which includes the
    endpoint; in Python this is usually not what you want.
```

Note: the name of a function can be used without the brackets, but this does not execute the function!

Try This Out

In your **console**, type in the following (run each line multiple times), and answer the questions below.

```
import random
print(random.randint(5, 20))        # line 1
print(random.randrange(3, 10, 2))   # line 2
print(random.uniform(2.5, 5.5))     # line 3
```

- *What did you see on line 1?*
  What was the smallest number you could have seen, what was the largest?

- *What did you see on line 2?*
  What was the smallest number you could have seen, what was the largest?
  Could line 2 have produced a 4?

- *What did you see on line 3?*
  What was the smallest number you could have seen, what was the largest?

Example to Study

This example combines some of the control structures (while loops, if statements) from last week with some useful string formatting for displaying currency.

Capitalist Conrad wants us to write a stock-price simulator for a volatile stock. The price starts off at $10.00, and, at the end of every day there is a 50% chance it increases by 0 to 10%, and a 50% chance that it decreases by 0 to 5%. If the price rises above $1000, or falls below $0.01, the program should end. The price should be displayed to the nearest cent (e.g. $33.59, not $33.5918232901).

What module do we need to import? random
What functions from random do we need to use? randint (for the 50% chance of increase or decrease), and uniform (to generate a random floating-point number)

Download the code from: https://github.com/CP1404/Practicals/blob/master/Prac02/capitalist_conrad.py
(Note that copying from PDFs is generally problematic as the spacing doesn't usually transfer correctly.)

```
import random
MAX_INCREASE = 0.1  # 10%
MAX_DECREASE = 0.05  # 5%
MIN_PRICE = 0.01
MAX_PRICE = 1000.0
INITIAL_PRICE = 10.0

price = INITIAL_PRICE
print("${:,.2f}".format(price))

while price >= MIN_PRICE and price <= MAX_PRICE:
    price_change = 0
    # generate a random integer of 1 or 2
```

```python
# if it's 1, the price increases, otherwise it decreases
if random.randint(1, 2) == 1:
    # generate a random floating-point number
    # between 0 and MAX_INCREASE
    price_change = random.uniform(0, MAX_INCREASE)
else:
    # generate a random floating-point number
    # between negative MAX_INCREASE and 0
    price_change = random.uniform(-MAX_DECREASE, 0)

price *= (1 + price_change)
print("${:,.2f}".format(price))
```

Things To Do:

1. The program currently runs without telling us how many days it simulated.
   Add this feature (use string formatting) so that the program prints like:

   ```
   Starting price: $10.00
   On day 1 price is: $9.89
   ...
   On day 424 price is: $915.71
   On day 425 price is: $1,001.60
   ```

2. Notice how the use of CONSTANTS makes the program easier to read and customise.
   Try changing these so the allowed price range is $1 to $100 and the increase could be up to 17.5%
   Run the program with these new values.

Exceptions

Here is some example code that uses exceptions, as discussed in lectures recently.
Copy the following code into a new file, and run it, then answer the questions below…

```python
try:
    numerator = int(input("Enter the numerator: "))
    denominator = int(input("Enter the denominator: "))
    fraction = numerator / denominator
except ValueError:
    print("Numerator and denominator must be valid numbers!")
except ZeroDivisionError:
    print("Cannot divide by zero!")
print("Finished.")
```

Questions

1. When will a ValueError occur?
2. When will a ZeroDivisionError occur?
3. Could you change the code to avoid the possibility of a ZeroDivisionError?

Intermediate Exercises

Computers use ASCII to define a character-encoding scheme for letters, digits, and other characters. It is useful to become familiar with ASCII since that is how string comparisons are made.

Mr. Black the school teacher wants an educational program for his school students to explore the details of ASCII. He wants the app to allow a student to input a character and see the corresponding ASCII code, and vice versa. The output of the program should look like (where **g** and **100** are user inputs):

```
Enter a character: g
The ASCII code for g is 103
Enter a number between 33 and 127: 100
The character for 100 is d
```

Things To Do:

1. Start new file, asciiTable.py, and write code for this program. Remember that you can use the ord() and chr() functions to convert characters to ASCII and vice versa.

2. Add error checking so that the number entered must be between the LOWER (33) and UPPER (127) bounds. Use constants for these values and use them in both your print statement and in your while loop condition. That is, the numbers 33 and 127 should appear only once.
   Use the **str.format()** method everywhere you print literals and variable values.

3. Now add on to the end of this program by writing code that displays a table with two columns, one for the numeric ASCII value and the other for the character itself. Use the **str.format()** method to align the text nicely in two columns. Print the values between 33 and 127. It should output like:

```
 33      !
 34      "
...
 99      c
100      d
...
```

Tips for string formatting with the format specifier {}
The part *before* the colon specifies which parameter to print. This can be left out to use the default order.
The part *after* the colon specifies the formatting. E.g. {:3} specifies to use 3 spaces (or more if needed) for the value when it's printed.
By default, numbers are right-aligned and strings are left-aligned. You can change this with > or <
So, {:>6} would format the value to be right-aligned and take up 6 (or more if needed) spaces.

Problem For You To Fill In The Blanks - Exceptions

Let's write a program that gets an integer from the user and does not crash when a non-number is entered.
Copy the code below and modify/add the parts highlighted so that it works and prints the number at the end:
https://github.com/CP1404/Practicals/blob/master/Prac02/exceptionsToComplete.py

```
finished = False
result = 0
while not finished:
    try:
        ▮▮▮▮▮▮
        ▮▮▮▮▮▮
    except ▮▮▮▮▮▮:
        print("Please enter a valid integer.")
print("Valid result is:", result)
```

Do-from-scratch Exercises

Files

Note: when you execute a Python program that contains a line like **open('data.txt', mode='w')** the new file "data.txt" is created in the same folder as the Python file in your PyCharm project.

1. Write a program that asks the user for their name, then opens a file called "name.txt" and writes that name to it.

2. Write a program that opens "name.txt" and reads the name (as above) then prints,
   "Your name is Bob" (or whatever the name is in the file).

3. Create a text file called "numbers.txt" (You can create a simple text file in PyCharm with Ctrl+N, choose "File" and save it in your project). Put the numbers 17 and 42 on separate lines in the file and save it:
   17
   42
   Write a program that opens "numbers.txt", reads the numbers and adds them together then prints the result, which should be… 59.
   Extended (perhaps only do this if you're cruising… if you are struggling, just read the solution):
   Now extend your program so that it can print the total for a file containing any number of numbers.

Password Checker

Download the starter code (complete with hints in the form of #TODO comments) from:
https://github.com/CP1404/Practicals/blob/master/Prac02/passwordChecker.py

Write a program that asks for and validates a person's password. The program is not for comparing a password to a known password, but validating the 'strength' of a new password, like you see on websites: enter your password and then it tells you if it's valid (matches the required pattern) and re-prompts if it's not. All passwords must contain *at least one each of: number*, *lowercase* and *uppercase* character.

The starter code uses constants (variables at the top of the code, named in ALL_CAPS) to store:
   a. the minimum and maximum length of the password
   b. whether or not a special character (not alphabetical or numerical) is required
When a valid password is entered, it should print it on the screen along with its length.

Output should look something like this:

```
Please enter a valid password
Your password must be between 5 and 15 characters, and contain:
    1 or more uppercase characters
    1 or more lowercase characters
    1 or more numbers
    and 1 or more special characters:  !@#$%^&*()_-=+`~,./'[]\<>?{}|
> this?
Invalid password!
> whyNot?CanIhaveThis?
Invalid password!
> 12345678901234567890aBcv@
Invalid password!
> thisISmy123Pass!
```

```
        Invalid password!
        > 1thisISit!
        Your 10 character password is valid: 1thisISit!
```

Here's another run with the same code but different values for the constants (special characters are not required in this version):

```
        Please enter a valid password
        Your password must be between 2 and 6 characters, and contain:
                1 or more uppercase characters
                1 or more lowercase characters
                1 or more numbers
        > aB
        Invalid password!
        > HowCanIHave2Chars?
        Invalid password!
        > 1aB
        Your 3 character password is valid: 1aB
```

Important Note: Do not just try and Google "Python password checker" or something, but think about doing this step by step. We have started this for you in the code provided.

- First, just check if a string has at least one lowercase character.
  You can do this by looping through the string (`for character in password:`) and testing each character… count the ones that match (**`character.islower()`**)… At the end you know how many lowercase characters there are.
- When you are able to count lowercase, then, in the same loop, count the uppercase characters
- Then, count the numbers…
  Test your code for each of these changes as you write them
- For special characters, remember you can use the **`in`** operator to see if the character is in another string (like a constant called SPECIAL_CHARACTERS)
- … keep going until you can tell how many of each kind of character there are
- Then put it all together and test with some different settings.

We hope this incremental approach makes sense and that you use it for everything you code.

When you have the program working, replace the inconsistent printing of text and variables with nice string formatting using the str.format() method.

Got your GitHub on?

If you haven't setup your own GitHub account, please do so now. See our instructions at:
https://github.com/CP1404/Starter/wiki/Software-Setup#github
Note that you should use a meaningful username that identifies who you are. JCU staff should be able to who you are from your username. Your GitHub account is an important and professional record of your work. You will likely use it as an online portfolio in the future.

Practice & Extension Work

The final, "extension" part of pracs will usually be for you to do outside of prac time.
Use these exercises as extra practice and as ways to learn new things.

Word Generator

The following program randomly generates words by constructing a string from random combinations of characters. The `word_format` variable stores a sequence like ccvc, which means: consonant consonant vowel consonant. The **random.choice** function is a useful way to select a single value from a sequence of values. Notice how the variable `word` starts as an empty string and then is constructed using repeated string concatenation (with the + operator).
Try this and see if you can get any interesting words:
Copy the code below from: https://github.com/CP1404/Practicals/blob/master/Prac02/wordGenerator.py

```python
import random

VOWELS = "aeiou"
CONSONANTS = "bcdfghjklmnpqrstvwxyz"

word_format = "ccvc"
word = ""
for kind in word_format:
    if kind == "c":
        word += random.choice(CONSONANTS)
    else:
        word += random.choice(VOWELS)
print(word)
```

Things To Do:
- Get the word format from the user so they can customise it. Convert it to lowercase using a str method.
- Try and make the program more interesting, For example:
    a. Use wildcards for the vowels (#) and consonants (%) or either (*) and make alphabetical characters use that actual character - e.g. the format "%re#*l*" might produce a word like "greatly" or "breuzla"
    b. Automatically (randomly) generate the word_format variable.

Automatic Password Generator

Write a program that should ask for a length and what characteristics it must have - requirements for upper/lower/numeric/special characters - then it should generate a password that matches.
Use your earlier program's checker functionality to validate the generated password.

ASCII Columns Challenge

Add columns to your ASCII table output. Ask the user for how many columns to print, then figure out how to write loop(s) and print statements to achieve this.

More Random Conrad

Replace the literal values for the constants at the top (like MAX_INCREASE) with randomly generated values (that make sense)

Solutions to File Exercises

1. Write a program that asks the user for their name, then opens a file called "name.txt" and writes that name to it.
2. Write a program that opens "name.txt" and reads the name (as above) then prints,
"Your name is Bob" (or whatever the name is in the file).
3. First, create a text file called "numbers.txt".
Put the numbers 17 and 42 on separate lines in the file and save it.
Write a program that opens "numbers.txt", reads the numbers and adds them together then prints the result, which should be… 59.

```python
# Quick Program 1
outFile = open("name.txt", "w")
name = input("What is your name? ")
print(name, file=outFile)  # or outFile.write(name)
outFile.close()


# Quick Program 2
in_file = open("name.txt", "r")
name = in_file.read().strip()
print("Your name is", name)
in_file.close()


# Quick Program 3
# Note that .strip() is unnecessary since int() handles that whitespace
in_file = open("numbers.txt", "r")
number1 = int(in_file.readline())
number2 = int(in_file.readline())
print(number1 + number2)
in_file.close()


# Quick Program 3 extended - sum of all numbers
in_file = open("numbers.txt", "r")
total = 0
for line in in_file:
    number = int(line)
    total += number
print(total)
in_file.close()
```