Weekend: Strings; Slicing; Functions and Exceptions

Strings

String Formatting

This is a way of embedding variables into strings. It is pretty easy to do.

```
name = "James"
age = 19

print(f"My name is {name} and I am {age} years old")
```

we run this

```
$ python3.6 example.py
My name is James and I am 19 years old
```

The `f` before the string stands for **f-strings** or formatted strings and allow us to put replacement fields `{}` with variable names inside our strings

Raw strings

A raw string is a special type of string that allows us to ignore all escape characters `\` and print them out

```
print('Beyonce\'s lemonade stand')
```

we run this

```
$ python example.py
Beyonce's lemonade stand
```

Here we see a proper use of the escape `\` character, by allowing us to add an apostrophe. But let us create a raw string and see what happens

```
print(r'Beyonce\'s lemonade stand')
```

we run this

```
$ python example.py
Beyonce\'s lemonade stand
```

We notice the `r` before the string. This means `raw` string. And in the output unlike the first example it prints out even the escape character.

Is x string methods

1. isalpha() - returns True if the string consists of letters only and is not blank
2. isalnum() - returns True if the string consists of numbers and letters and is not blank
3. isdecimal() - returns True if the string contains only numeric characters
4. isspace() - returns True if the string contains only space,tabs or new lines
5. istitle() - returns True if the string contains words that start with uppercase letters

```
alpha = "I like old music"
password = "K34jndnks"
number_string = "12345"
tabbs = "       "
titles = "I Love Cups"
false_titles = "I love Cups"

print( alpha.isalpha() )
print( password.isalnum() )
print( number_string.isdecimal() )
print( tabbs.isspace() )
print( titles.istitle() )
print( false_titles.istitle())
```

When we run this

```
$ python3.6 example.py
True
True
True
True
True
False
```

The last print statement was false because not all the words in `false_titles` were in uppercase.

These `xstring` methods are really useful when you want to do something like password validation.

String Exercises

1. **Mad Libs**

Create a mad libs kind of program that allows users to input word descriptions such as a "noun", a "verb", an "adjective" etc and create then display a funny story from the users input

Type casting

Type casting is the act of converting one type of data to another, so as to manipulate it in some way. Let's see an example of this.

```
# print out user's name and age
name = "James"
age = 19

print("My name is " + name + " I am " + age +" years old")
```

Let us run this code and see what happens.

```
$ python3.6 example.py

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
```

Here we see we get a `TypeError`, because python cannot concatenate a string with an integer. To use the `age` variable we need to convert it to a string.

```
# print out u users name and age
name = "James"
age = 19

print("My name is " + name + " I am " + str(age) + " years old")
```

Let us run this code and see what happens.

```
$ python3.6 example.py

My name is James I am 19 years old
```

In this example we enclose the `age` variable in a function called `str()`. The `str()` function converts the `age` from an integer to a string.

There are other type conversion methods:

1. `int()` converts what is passed to an integer.
2. `float()` converts what is passed to a float.

Let us see more examples of this

```
name = "James"
age = 19
weight = '79' # Kilograms

age_weight_ratio = int(weight)/age

print(age_weight_ratio)
```

Run this

```
$ python3.6 example.py
4.157894736842105
```

Here we converted string `weight`, to an integer and divided it by `age`.

Slicing

Slicing is the act of getting subsets or parts of strings, lists or tuples.

Let's get "slicing".

```
greetings = 'Hello, Moringa!'

part_one = greetings[0:5]
print(part_one)
```

When we run this we get:

```
$ python3.6 example.py
Hello
```

A slice statement is enclosed with `[]` square brackets and has two parts first is the position where to start slicing and second is the position to stop but not including that position.

`[start:stop]`

In the example above, we sliced the `greetings` variables, took the characters in the first 5 indexes and stored them in the variable `part_one`

You can also use negative indexes when slicing.

```
greetings = 'Hello, Moringa!'

part_one = greetings[-8:-1]
print(part_one)
```

When we run this we get:

```
$ python3.6 example.py
Moringa
```

In this code we start slicing counting backwards and get `Moringa` as our output

```
greetings = 'Hello, Moringa!'

part_one = greetings[0:]
print(part_one)
```

When we run this we get:

```
$ python3.6 example.py
Hello, Moringa!
```

We can also choose not to specify the start or end point of our slices. In this example, Python assumes that we want to slice from index `0`, to the last item and automatically fills in that value.

We can also slice lists

```
number = [1,2,3,4,5,6,7,8,9]
four_digits = number[:4]
print(four_digits)
```

When we run this we get:

```
$ python3.6 example.py
[1,2,3,4]
```

Here we did not specify the beginning of our slice and python automatically filled that in to be at index `0`.

Functions

We have already used functions to perform several tasks like `range` and `print` but let's learn how to create our own functions in python.

Functions are blocks of code that begin with the `def` keyword

```
def fun_a():
    print("I have been called")

fun_a()

"I have been called"
```

This example we have created a function with no arguments and in the function we have a print statement that outputs a string

We then went on to call the function and we got our desired output.

Passing Arguments

In python you can also pass arguments to functions

```
def fun_a (a,b):
    print(a+b)

fun_a(1,4)
5
```

Python can also take on **keyword arguments**. These are arguments that are already defined

```
def fun_a(a = 1,b = 4):
    print(a+b)

fun_a(a=6,b=7);
13
```

Here, we created a function with two `keyword arguments`. When we call the function we changed the values of those arguments to our own values

We could also call the function without passing any parameters. Then the default keyword arguments values will be passed in

```
def fun_a(a = 1,b = 4):
    print(a+b)

fun_a();
5
```

Creating an Empty function

Sometimes we might want to define a function and not put code in it. This could be for whatever reason like outlining your code.

```
# Empty function
def fun_a():
```

If we run our code like this, it will give us an error. Python has a keyword called **pass** that can help us here. `pass` allows us to create empty blocks of code because when it runs it returns a null or nothing.

```
# Empty function
def fun_a():
    pass
```

Now when we run our code it will work perfectly.

Functions that return something

Functions can also return values to us

```
def fun_a (a,b):
    return a+b

sum = fun_a(4,5)

print(sum)
9
```

Here, we use the `return` statement, to get the value from the operation performed in the function.

Exceptions and Error Handling

In technical terms **Exception Handling** is the mechanism for stopping normal program flow and continuing it at some surrounding code block.

Simply put it is the art of spotting fatal errors that may come up in our applications and handling those errors.

Right now an error breaks our application.

create a new Folder called Error-Handling and inside create a file `handle.py`

```
def get_age():
    print("How old are you ")
    age = int(input())
    return age

print(get_age())
```

Let's take this simple application for example where we get the age of a user

```
$ python3.6 handle.py
How old are you
19
19
```

The program works fine if we enter a number. But what if a user enters a name

```
$ python3.6 handle.py
How old are you
nineteen
```

```
Traceback (most recent call last):
  File "testapp.py", line 6, in <module>
    input_age = get_age()
  File "testapp.py", line 3, in get_age
    age = int(input())
ValueError: invalid literal for int() with base 10: 'age'
```

We get a `ValueError` because we put in a non integer string and the `int()` constructor cannot convert the string to integer.

This error is difficult to understand. Let us create a more understandable error message

```
def get_age():
    print("How old are you ")
    try:
        age = int(input())
        return age
    except ValueError:
        return "That was not a valid input"
```

We use the `try\except` block to handle errors. Inside the `try` block we put in code that may throw an Error and in the `except` block we catch the thrown error and provide code to handle that error.

Programmer Errors

There are some error that are caused by programmers themselves. These errors should not be handled but fixed.

1. `IndentationError` - when you fail to separate code blocks properly.
2. `NameError` - when you call an undefined variable function or method.
3. `TypeError` - when you try and perform operations on unrelated types .