COMPUTER PROGRAMMING
LEARNING GUIDE 10

Revenge of the Strings

Here's a cool trick that can be done with strings:

```
def shout(string):
for character in string:
        print("Gimme a " + character)
        print("'" + character + "'")

shout("Lose")

def middle(string):
        print("The middle character is:", string[len(string) // 2])

middle("abcdefg")
middle("The Python Programming Language")
middle("Frances Kelsey")
```

What these programs demonstrate is that strings are similar to lists in several ways. The `shout()` function shows that `for` loops can be used with strings just as they can be used with lists. The `middle` procedure shows that strings can also use the `len()` function and array indexes and slices. Most list features work on strings as well.

The next program demonstrates some string specific features:

```
def to_upper(string):
        ## Converts a string to upper case
        upper_case = ""
        for character in string:
                if 'a' <= character <= 'z':
                        location = ord(character) - ord('a')
                        new_ascii = location + ord('A')
                        character = chr(new_ascii)
                upper_case = upper_case + character
        return upper_case

print(to_upper("This is Text"))
```

This works because the computer represents the characters of a string as numbers from 0 to 1,114,111. For example 'A' is 65, 'B' is 66 and א is 1488. The values are the unicode value. Python has a function called `ord()` (short for ordinal) that returns a character as a number. There is also a corresponding function called `chr()` that converts a number into a character. With this in mind the program should start to be clear. The first detail is the line: `if 'a' <= character <= 'z':` which checks to see if a letter is lower case. If it is then the next lines are used. First it is converted into a location so that `a = 0, b = 1, c = 2` and so on with the line: `location = ord(character) - ord('a')`. Next the new value is found with `new_ascii = location + ord('A')`. This value is converted back to a character that is now upper case. Note that if you really need the upper case of a letter, you should use `u=var.upper()` which will work with other languages as well.

Now for some interactive typing exercise:

```
>>> # Integer to String
>>> 2
2
>>> repr(2)
'2'
>>> -123
-123
>>> repr(-123)
'-123'
>>> # String to Integer
>>> "23"
'23'
>>> int("23")
23
>>> "23" * 2
'2323'
>>> int("23") * 2
46
>>> # Float to String
>>> 1.23
1.23
>>> repr(1.23)
'1.23'
>>> # Float to Integer
>>> 1.23
1.23
>>> int(1.23)
1
>>> int(-1.23)
-1
>>> # String to Float
>>> float("1.23")
1.23
>>> "1.23"
'1.23'
>>> float("123")
123.0
```

If you haven't guessed already the function `repr()` can convert an integer to a string and the function `int()` can convert a string to an integer. The function `float()` can convert a string to a float. The `repr()` function returns a printable representation of something. Here are some examples of this:

```
>>> repr(1)
'1'
>>> repr(234.14)
'234.14'
>>> repr([4, 42, 10])
'[4, 42, 10]'
```

The `int()` function tries to convert a string (or a float) into an integer. There is also a similar function called `float()` that will convert a integer or a string into a float. Another function that Python has is the `eval()` function. The `eval()` function takes a string and returns data of the type that Python thinks it found.

For example:

```
>>> v = eval('123')
>>> print(v, type(v))
123 <type 'int'>
>>> v = eval('645.123')
>>> print(v, type(v))
645.123 <type 'float'>
>>> v = eval('[1, 2, 3]')
>>> print(v, type(v))
[1, 2, 3] <type 'list'>
```

If you use the `eval()` function, you should check that it returns the type that you expect. One useful string function is the `split()` method. Here's an example:

```
>>> "This is a bunch of words".split()
['This', 'is', 'a', 'bunch', 'of', 'words']
>>> text = "First batch, second batch, third, fourth"
>>> text.split(",")
['First batch', ' second batch', ' third', ' fourth']
```

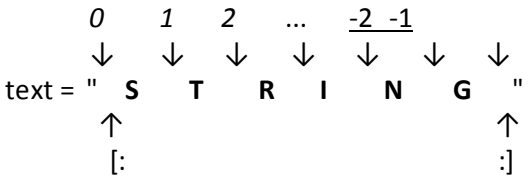
Notice how `split()`converts a string into a list of strings. The string is split by whitespace by default or by the optional argument (in this case a comma). You can also add another argument that tells `split()`how many times the separator will be used to split the text. For example:

```
>>> list = text.split(",")
>>> len(list)
4
>>> list[-1]
' fourth'
>>> list = text.split(",", 2)
>>> len(list)
3
>>> list[-1]
' third, fourth'
```

Slicing strings (and lists)

Strings can be cut into pieces — in the same way as it was shown for lists in a previous section — by using the *slicing* "operator" `[]`. The slicing operator works in the same way as before: `text[first_index:last_index]` (in very rare cases there can be another colon and a third argument, as in the example shown below).

In order not to get confused by the index numbers, it is easiest to see them as *clipping places*, possibilities to cut a string into parts. Here is an example, which shows the clipping places (arrows) and their index numbers (italics and underlined) for a simple text string:



Note that the italic indexes are counted from the beginning of the string and the underlined ones from the end of the string backwards. (Note that there is no underlined -0, which could seem to be logical at the end of the string. Because `-0 == 0`, `-0` means "beginning of the string" as well.)

Now we are ready to use the indexes for slicing operations:

t	→	"
t	→	"
t	→	"
t	→	"
t	→	"
t	→	"
t	→	"

`text[1:4]` gives us all of the `text` string between clipping places 1 and 4, `"TRI"`. If you omit one of the [first_index:last_index] arguments, you get the beginning or end of the string as default: `text[:5]` gives "STRIN". For both `first_index` and `last_index` we can use both the italic and the underlined blue numbering schema: `text[:-1]` gives the same as `text[:5]`, because the `index -1` is at the same place as `5` in this case. If we do not use an argument containing a colon, the number is treated in a different way: `text[2]` gives us one character following the second clipping point, "R". The special slicing operation `text[:]` means "from the beginning to the end" and produces a copy of the entire string (or list, as shown previously).

Last but not least, the slicing operation can have a second colon and a third argument, which is interpreted as the "step size": `text[::-1]` is `text` from beginning to the end, with a step size of -1. -1 means "every character, but in the other direction". `"STRING"` backwards is `"GNIRTS"` (test a step length of 2, if you have not got the point here).

All these slicing operations work with lists as well. In that sense strings are just a special case of lists, where the list elements are single characters. Just remember the concept of *clipping places*, and the indexes for slicing things will get a lot less confusing.

Examples

```
# This program requires an excellent understanding of decimal numbers.
def to_string(in_int):
        """Converts an integer to a string"""
        out_str = ""
        prefix = ""
        if in_int < 0:
                prefix = "-"
                in_int = -in_int
        while in_int // 10 != 0:
                out_str = str(in_int % 10) + out_str
                in_int = in_int // 10
                out_str = str(in_int % 10) + out_str
        return prefix + out_str

def to_int(in_str):
        """Converts a string to an integer"""
        out_num = 0
        if in_str[0] == "-":
                multiplier = -1
                in_str = in_str[1:]
        else:
                multiplier = 1
        for c in in_str:
                out_num = out_num * 10 + int(c)
        return out_num * multiplier

print(to_string(2))
print(to_string(23445))
print(to_string(-23445))
print(to_int("14234"))
print(to_int("12345"))
print(to_int("-3512"))
```

File IO

Here is a simple example of file I/O (input/output):

```
# Write a file
with open("test.txt", "wt") as out_file:
        out_file.write("This Text is going to out file\nLook at it and see!")

# Read a file
with open("test.txt", "rt") as in_file:
        text = in_file.read()

print(text)
```

Notice that it wrote a file called test.txtin the directory that you ran the program from. The ₩nin the string tells Python to put a *n*ewline where it is.

A overview of file I/O is:

- Get a file object with the openfunction
- Read or write to the file object (depending on how it was opened)
- If you did not use withto open the file, you'd have to close it manually

The first step is to get a file object. The way to do this is to use the openfunction. The format is `file_object = open(filename, mode)` where `file_object` is the variable to put the file object, `filename` is a string with the filename, and `mode` is "rt"to *r*ead a file as *t*ext or "wt"to *w*rite a file as *t*ext (and a few others we will skip here). Next the file objects functions can be called. The two most common functions are `read` and `write`. The write function adds a string to the end of the file. The readfunction reads the next thing in the file and returns it as a string. If no argument is given it will return the whole file (as done in the example).

Now here is a new version of the phone numbers program that we made earlier:

```
def print_numbers(numbers):
        print("Telephone Numbers:")
         for k, v in numbers.items():
         print("Name:", k, "\tNumber:", v)
print()

def add_number(numbers, name, number):
        numbers[name] = number

def lookup_number(numbers, name):
        if name in numbers:
        return "The number is " + numbers[name]
else:
        return name + " was not found"

def remove_number(numbers, name):
        if name in numbers:
                del numbers[name]
        else:
                print(name," was not found")

def load_numbers(numbers, filename):
        in_file = open(filename, "rt")
        while True:
                in_line = in_file.readline()
                if not in_line:
                        break
                in_line = in_line[:-1]
                name, number = in_line.split(",")
                numbers[name] = number
        in_file.close()

def save_numbers(numbers, filename):
        out_file = open(filename, "wt")
        for k, v in numbers.items():
                out_file.write(k + "," + v + "\n")
        out_file.close()

def print_menu():
        print('1. Print Phone Numbers') print('2. Add a Phone Number') print('3. Remove a Phone
        Number') print('4. Lookup a Phone Number') print('5. Load numbers')
        print('6. Save numbers')
        print('7. Quit')
        print()
```

THAT COMPLETES THE DEFINITIONS

THE REST OF THE PROGRAM IS ON THE NEXT PAGE

```
phone_list = {}
menu_choice = 0

print_menu()
while True:
        menu_choice = int(input("Type in a number (1-7): "))
        if menu_choice == 1:
                print_numbers(phone_list)
        elif menu_choice == 2:
                print("Add Name and Number")
                name = input("Name: ")
                phone = input("Number: ")
                add_number(phone_list, name, phone)
        elif menu_choice == 3:
                print("Remove Name and Number")
                name = input("Name: ")
                remove_number(phone_list, name)
        elif menu_choice == 4:
                print("Lookup Number")
                name = input("Name: ")
                print(lookup_number(phone_list, name))
        elif menu_choice == 5:
                filename = input("Filename to load: ")
                load_numbers(phone_list, filename)
        elif menu_choice == 6:
                filename = input("Filename to save: ")
                save_numbers(phone_list, filename)
        elif menu_choice == 7:
                break
        else:
                print_menu()

print("Goodbye")
```

Notice that it now includes saving and loading files.

First we will look at the save portion of the program. First it creates a file object with the command `open(filename, "wt")`. Next it goes through and creates a line for each of the phone numbers with the command `out_file.write(k + "," + v + "\n")`. This writes out a line that contains the name, a comma, the number and follows it by a newline.

The loading portion is a little more complicated. It starts by getting a file object. Then it uses a `while True:` loop to keep looping until a breakstatement is encountered. Next it gets a line with the line `in_line = in_file.readline()`. The `readline`function will return a empty string when the end of the file is reached. The if statement checks for this and breaks out of the whileloop when that happens. Of course if the `readline` function did not return the newline at the end of the line there would be no way to tell if an empty string was an empty line or the end of the file so the newline is left in what `readline`returns. Hence we have to get rid of the newline. The line `in_line = in_line[:-1]` does this for us by dropping the last character. Next the line name, `number = in_line.split(",")` splits the line at the comma into a name and a number. This is then added to the `numbers`dictionary.

Advanced use of .txt files

You might be saying to yourself, "Well I know how to read and write to a textfile, but what if I want to print the file without opening out another program?"

There are a few different ways to accomplish this. The easiest way does open another program, but everything is taken care of in the Python code, and doesn't require the user to specify a file to be printed. This method involves invoking the subprocess of another program.

Remember the file we wrote output to in the above program? Let's use that file. Keep in mind, in order to prevent some errors, this program uses concepts from the section that follows this. Please feel free to revisit this example after the next section.

```
import subprocess
def main():
    try:
        print("This small program invokes the print function in the Notepad application")
        #Lets print the file we created in the program above
        subprocess.call(['notepad','/p','numbers.txt'])
    except WindowsError:
        print("The called subprocess does not exist, or cannot be called.")

main()
```

The `subprocess.call` takes three arguments. The first argument in the context of this example, should be the name of the program which you would like to invoke the printing subprocess from. The second argument should be the specific subprocess within that program. For simplicity, just understand that in this program, '`/p`'is the subprocess used to access your printer through the specified application. The last argument should be the name of the file you want to send to the printing subprocess. In this case, it is the same file used earlier.

Exercises

Modify the grades program from section Dictionaries so that is uses file I/O to keep a record of the students.

Dealing with the imperfect

...or how to handle errors closing files with with

We use the "with" statement to open and close files.

```
with open("in_test.txt", "rt") as in_file:
    with open("out_test.txt", "wt") as out_file:
        text = in_file.read()
        data = parse(text)
        results = encode(data)
        out_file.write(results)
print( "All done." )
```

If some sort of error happens anywhere in this code (one of the files is inaccessible, the parse() function chokes on corrupt data, etc.) the "with" statements guarantee that all the files will eventually be properly closed. Closing a file just means that the file is "cleaned up" and "released" by our program so that it can be used in another program.

catching errors with try

So you now have the perfect program, it runs flawlessly, except for one detail, it will crash on invalid user input. Have no fear, for Python has a special control structure for you. It's called `try` and it tries to do something. Here is an example of a program with a problem:

```
print("Type Control C or -1 to exit")
number = 1
while number != -1:
    number = int(input("Enter a number: "))
    print("You entered:", number)
```

Notice how when you enter @#& it outputs something like:

```
Traceback (most recent call last):
File "try_less.py", line 4, in <module>
number = int(input("Enter a number: "))
ValueError: invalid literal for int() with base 10: '\\@#&'
```

As you can see the `int()` function is unhappy with the number `@#&` (as well it should be). The last line shows what the problem is; Python found a `ValueError`. How can our program deal with this? What we do is first: put the place where errors may occur in a `try` block, and second: tell Python how we want `ValueErrors` handled. The following program does this:

```
print("Type Control C or -1 to exit")
number = 1
while number != -1:
    try:
    number = int(input("Enter a number: "))
print("You entered:", number)
    except ValueError:
print("That was not a number.")
```

Now when we run the new program and give it @#& it tells us "That was not a number." and continues with what it was doing before.

When your program keeps having some error that you know how to handle, put code in a `try` block, and put the way to handle the error in the `except` block.

Exercises

Update at least the phone numbers program (in section Dictionaries) so it doesn't crash if a user doesn't enter any data at the menu.