Software skills for librarians

Module 3: Programming in Python Answers

1c. This gives an error message, for example:

```
File "hellow.py", line 2
  print("Hello World!)
                     ^
SyntaxError: EOL while scanning string literal
```

	Advantages	Disadvantages
Interactively	Get instant feedback Easier to try things out	Difficult for long programs Or frequently used programs
Batch	Easier to repeat programs Convenient for long programs Better editing facilities	More difficult to debug No feedback on correct syntax

2. Too many to list, but likely answers include Ada, Assembler, Basic, BCPL, C, C++, Fortran, Java, JavaScript, LISP, ObjectiveC, Pascal, Perl, Python.

3. The following answers all three sections:

```
username=raw_input("Your name? ")
print "Hello World! from", username
username="Alan Turing"
print "username is now ", username
```

4. Parts a to c can be solved as follows:

```
authors=["Dickens, Charles", "Hardy, Thomas", "Austen, Jane",
"Bronte, Charlotte"]
authors.sort()
authors.reverse()
print authors
newlist=authors[1:3]
print newlist
```

4d. No, because tuples are immutable; they cannot be sorted or reversed in place.

5. Parts a and b:

```
fieldsused=myrecord.keys()
title=myrecord["245"]
```

5b. A string: `print type(title)` gives `<type 'str'>`
5d. Put each field in a list or tuple with two elements, the indicators and the field data.

6. The following is a model answer, although it is not strictly necessary to be case insensitive:

```
lang=raw_input("Whats your favourite programming language? ")
if lang=="Python" or lang=="python":
    print("Good choice!")
else:
    print("Have you thought of using Python?")
```

7. This is a case of adding an extra clause to the if statement:

```
elif lang=="Java" or lang=="java":
    print("Its a modern object-oriented language.")
```

8. The difficulty is knowing when to terminate the input loop. the input() function raises an error for a blank like so its easiest to use an input of zero to mark the end. Also, in this case, it is more helpful to keep the loop test at the end, which requires an explicit break in Python:

```
numbers=[]
while True:
    n=input()
    if n==0: break
    numbers.append(n)
sum=0
for n in numbers:
    sum=sum+n
print "The average is ", sum/len(numbers)
```

8b. The version with only one loop needs to form the sum and count the numbers as they are entered:

```
sum=0
nums=0
while True:
    n=input()
    if n==0: break
    nums=nums+1
    sum=sum+n
print "The average is ", sum/nums
```

8c. The second version should be faster because operations like addition are very fast in comparison with the overheads of a loop.

9. This is a case of enclosing the existing code in a function definition. Another solution would be returning true or false explicitly in another if statement:

```
def valid_isbn(isbn):
    t=0
    for i in range(0, 9):
        n=int(isbn[i])
        t=t+(i+1)*n
    check=t%11
    if check==10:
        check="X"
    else:
        # Convert number to string
        check="%1d" % check
    return(check==isbn[9])

print valid_isbn("9810240201")
print valid_isbn("0596158064")
```

10. As before, but enclosing the function in a class definition, and accessing the ISBN number via self:

```
class isbn:
    def __init__(self, isbn):
        self.isbn=isbn

    def validate(self):
        t=0
        for i in range(0, 9):
            n=int(self.isbn[i])
            t=t+(i+1)*n
        check=t%11
        if check==10:
            check="X"
        else:
            # Convert number to string
            check="%1d" % check
        return(check==self.isbn[9])

my_isbn=isbn("9810240201")
print my_isbn.validate()
print isbn("0596158064").validate()
```

11a. Similar to before, but we need to open a file and the input function is different, so this version takes advantage of an iterator. The conversion from string to number is performed explicitly using float():

```
sum=0
nums=0
with open("numbers.txt", "r") as fh:
    for line in fh:
        if line!="":
            sum=sum+float(line)
            nums=nums+1
print "The average is ", sum/nums
```

11b. Either using separate loops to read the numbers into a list and add them up; read the numbers and add them in a single loop as above; or use the readlines() method and a loop to add the numbers.

12. Add the following lines to the end of marcdict.py:

```
fields=myrecord.keys()
fields.sort()
with open("newrecord.txt", "w") as fh:
    for tag in fields:
        field=tag+" __ "+myrecord[tag]
        fh.write(field)
```

13. The class definition and `__init__()` method are straightforward, just a case of defining an empty dictionary:

```
class marcrecord:
    def __init__(self):
        self.recdata={}
```

The trick with the subscript notation is to realise that the dictionary keys will be strings, and the user will likely use numbers in the square brackets, so we use the format string trick to convert a number to a string:

```
    def __getitem__(self, index):
        ftag="%03d" % index
        return self.recdata[ftag]
```

Reading and writing a record is much the same as before, except that we have to handle repeated fields. This implementation puts the field data in a list, so we have a dictionary of lists. Thus, for example, fetching field 500 using the subscript notation, actually returns a list of all notes.

```
    def readrec(self, fh):
        for field in fh:
            tag=field[0:3]
            fdata=field[7:]
            # Use lists to handle repeated fields
            if tag not in self.recdata.keys():
                mylist=[]
                mylist.append(fdata)
                self.recdata[tag]=mylist
            else:
                self.recdata[tag].append(fdata)

    def writerec(self, fh):
        fields=self.recdata.keys()
        fields.sort()
        # Write fields in numerical order
        for tag in fields:
            flist=self.recdata[tag]
            for fdata in flist:
                field=tag+" __ "+fdata
                fh.write(field)
```