Software skills for librarians

Module 3: Programming in Python
Reference material

Control structures

The if statement runs a block of statements only if the specified condition is true, otherwise the optional else statement is executed. The test condition can be anything which produces a boolean `True` or `False` result. You can use comparisons like <, >, <=, >=, != and ==. Several comparisons can be combined with `and`, and `or`.

```
if condition:
    statement block
else:
    statement block
```

There are two basic forms of loop, the `for` loop iterates over a known set of items, the `while` loop continues for as long as the specified condition remains true. An infinite loop can be specified by writing `while True:`

```
for index in list:
    statement block

while condition:
    statement block
```

Functions are defined with the `def` keyword, followed by a list of the parameters they expect. The values given when calling the function are copied into the named parameters. Any changes made to these do not affect the corresponding values outside of the function.

```
def fn_name(parameters):
    statement block
    return value
```

Several functions which operate on the same type of data can be packaged together in a `class`, in which case they are known as methods. An instance of a class is called an object. You can define the special method called `__init__` which is called every time an object is created. The basic format of a class looks like:

```
class object:
    def __init__(self, parameters):
        initialisation code

    def method(parameters):
        statement block
```

When an error occurs it is useful to be able to raise an exception, which signals that something has gone wrong, and allows your code to take appropriate action, or to recover. You can do this with the `raise` statement. The exception can be detected with `try,` and handled with the `except` statements. The `finally` block is always executed regardless of any error:

```
try:
    statement
except errorname:
    statement block
finally:
    statement block
```

Operations on strings

`str="abc"`
Initialise a string.

`S=str[i]`
Return the ith character from the string, counting from 0.

`S=str[i:j]`
Return characters i to j from the string.

`str=S1 + S2`
Concatenate two strings.

`i=str.find(`*sub, start, end*`)`
Find the first occurence of substring sub within the string and return its position. If the optional parameters start and end are specified, search the slice s[start:end] instead. Return -1 if sub is not found.

`s=str.lower()`
Return a copy of the string with all characters converted to lowercase.

`s=str.replace(`*old, new, count*`)`
Return a copy of the string with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

`list=str.split(`*sep, maxsplit*`)`
Return a list of the words in the string, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done (thus, the list will have at most maxsplit+1 elements). If maxsplit is not specified or -1, then there is no limit on the number of splits (all possible splits are made).

`s=str.strip(`*chars*`)`
Return a copy of the string with the leading and trailing characters removed. The chars argument is a string specifying the set of characters to be removed. If this is omitted the default is to remove whitespace.

`s=str.upper()`
Return a copy of the string with all the characters converted to uppercase.

Operations on lists

`L=[1, 2, 3]`
Initialise a list of values.

`L=list(range(1,10))`
Initialise with a list of successive values.

`val=list[i]`
Return item i.

`L=list[i:j]`
Return a list containing items i to j.

`list=L1 + L2`
Concatenate two lists.

`list.append(`*x*`)`
Append x to the end of the list.

`list.insert(`*i, x*`)`
Insert x into the list in position i.

`x=list.pop(`*i*`)`
Remove the item at position i and return it.

`list.reverse()`
Reverse the order of the list.

`list.sort()`
Sort the list into order.

Operations on dictionaries

`dict={key: value}`
Initialise a dictionary with a list of key/value pairs.

`val=dict[key]`
Indexing by key.

`dict[key]=val`
Add or change the value associated with a key.

`key in dict`
Test for the presence of a key.

`l=dict.keys()`
Return a list of keys in the dictionary.

`i=len(`*dict*`)`
Return the number of key/value pairs in the dictionary.

`i=dict.pop(`*key*`)`
Remove key from the dictionary and return its value.

`l=dict.values()`
Return a list of all values in the dictionary.

Operations on files

`file.close()`
Close the file so that it cannot be read or written any more.

`file=open(`*name, mode*`)`
Open a named file and return a file object. If the file cannot be opened, IOError is raised. The most commonly used values of mode are 'r' for reading and 'w' for writing.

`s=file.readline(`*size*`)`
Read one entire line from the file. A trailing newline character is kept in the string (but may be absent when a file ends with an incomplete line). If the size argument is present

`l=file.readlines(`*sizehint*`)`
Read until EOF using readline() and return a list containing the lines thus read.

`file.seek(`*offset*`)`
Set the file's current position.

`i=file.tell()`
Return the file's current position.

`file.write(`*str*`)`
Write a single string to the file.

`file.writelines(`*list*`)`
Write a list of strings to the file.