Loops, Conditional Statements and Functions in Python

Quantitative Applications for Data Analysis

Alexey Fedoseev

March 5, 2024

Conditional statements

`if` statement evaluates whether a statement is true or false, and run code only in the case that the statement is true.

```
>>> temperature = -15
>>> if temperature < -10:
...     print("It is cold!")
...
It is cold!
```

If you want the program to do something even when an if statement evaluates to false use `else` statement.

```
>>> if temperature < -10:
...     print("It is cold!")
... else:
...     print("It is warm")
...
It is cold!
```

Else if statement

In many cases, we will want a program that evaluates more than two possible outcomes. For this, we will use an else if statement `elif`

```
>>> if temperature < -10:
...     print("It is cold!")
... elif (temperature > -10) and (temperature < 10):
...     print("It is mild")
... else:
...     print("It is warm")
...
It is cold!
```

Code blocks in Python start with ":" and are preceded by white spaces. The amount of indentation indicates which code block it's part of. Different code blocks can have different amounts of indentation, but the indentation must be consistent within the same code block.

for loops

A `for` loop implements the repeated execution of code based on a loop counter or loop variable. This means that `for` loops are used most often when the number of iterations is known before entering the loop, unlike `while` loops which are conditionally based.

```
>>> shopping_list = ["apples", "oranges", "grapes", "tomatoes"]
>>> for item in shopping_list:
...     print("I need to buy", item)
...
I need to buy apples
I need to buy oranges
I need to buy grapes
I need to buy tomatoes
```

Notice that code blocks in Python are specified using tabulation. It means that the code inside loops **must** be indented.

Nested code blocks

```
>>> temperature = -5
>>> shopping_list = ["apples", "oranges", "grapes", "tomatoes"]
>>> if temperature > -10:
...     print("I am going out!")
...     for item in shopping_list:
...             print("I need to buy", item)
... else:
...     print("I am staying home!")
...
I am going out!
I need to buy apples
I need to buy oranges
I need to buy grapes
I need to buy tomatoes
```

range

Python's built-in `range()` function is handy when you need to perform an action a specific number of times. It generates the integer numbers between the given start integer to the stop integer, which is generally used to iterate over with a `for` loop.

```
>>> len(shopping_list)
4
>>> range(len(shopping_list))
range(0, 4)
>>> for index in range(len(shopping_list)):
...     print("I need to buy", shopping_list[index])
...
I need to buy apples
I need to buy oranges
I need to buy grapes
I need to buy tomatoes
```

while loops

A `while` loop implements the repeated execution of code based on a given boolean condition. The code that is in a `while` block will execute as long as the `while` statement evaluates to `True`.

You can think of the `while` loop as a repeating conditional statement. After an if statement, the program continues to execute code, but in a while loop, the program jumps back to the start of the while statement until the condition is `False`.

```
>>> temperature = 2
>>> while temperature > -2:
...     print("Temperature is", temperature)
...     temperature = temperature - 1
...
Temperature is 2
Temperature is 1
Temperature is 0
Temperature is -1
```

Defining functions

A function is a block of instructions that performs an action and, once defined, can be reused. Functions make code more modular, allowing you to use the same code over and over again.

Let us create a file named `temperature.py` and define a function `check_temp`.

```
def check_temp(temperature):
    if temperature < -10:
        print("It is cold")
    elif temperature < 10:
        print("It is mild")
    elif temperature < 25:
        print("It is warm")
    else:
        print("It is hot!")

check_temp(-5)
check_temp(15)
```

As with all code blocks the body of a function starts with a colon and must be indented.
A function can take arguments. They are listed the same way as in R. When called, that which is passed to the function is assigned to the variable declared in the function definition.
That means, in case of our function `check_temp`, the value of `-5` is given to the variable `temperature`.
Run the script `temperature.py` to see the output.

```
$ python temperature.py
It is mild
It is warm
```

Default values

As with R, functions can take multiple arguments.

```
def check_temp(temperature = 0):
        if temperature < -10:
                print("It is cold")
        elif temperature < 10:
                print("It is mild")
        elif temperature < 25:
                print("It is warm")
        else:
                print("It is hot!")

check_temp()
check_temp(15)
```

- All non-optional arguments must be specified when a function is called.
- Notice that the argument values are assigned in the order they appear in the function declaration.
- The values of optional arguments are specified in the function definition.

```
$ python temperature.py
It is mild
It is warm
```

Global and local variables

Everything you were taught about global and local variables in the R classes applies to Python as well.

- Variables which are declared within a function are called "local". They may only be accessed within the function.
- Variables which are declared outside of functions are called "global".
  - Global variables can be accessed from within functions, but this is a bad idea.
  - It's better to pass all information you need in your function as an argument.
- If you start using global variables within functions you will break the modularity of your code. Your code will become less portable and much harder to debug.

return statement

You can pass a parameter value into a function, and a function can also produce a value.

Unlike with R, in Python you must include a return statement with your functions if you want to return something.

Create `listUtils.py`:

```
def list2str(lst):
    result = []
    for elem in lst:
        result.append(str(elem))
    return(result)

print(list2str([1,2,3,4]))
```

Run `listUtils.py`

```
$ python listUtils.py
['1', '2', '3', '4']
```

Importing modules

Writing a module is just like writing any other Python file. Modules can contain definitions of functions, classes, and variables that can then be utilized in other Python programs.

Let us say that we put several useful functions into the file `listUtils.py`. In order to use the function this file we need to `import` them first.

```
$ python
>>> import listUtils
['1', '2', '3', '4']
>>> listUtils.list2str([5,4,3,2,1])
['5', '4', '3', '2', '1']
```

Finding your functions

Let us say I moved my utility script to the directory `/Users/alexey/code`. We need to point Python to the new directory.

```
$ python
>>> import listUtils
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'listUtils'
>>> import sys
>>> sys.path
['/Users/alexey/anaconda3/lib/python3.6', ...,
'/Users/alexey/anaconda3/lib/python3.6/site-packages']
>>> sys.path.append('/Users/alexey/code')
>>> import listUtils
['1', '2', '3', '4']
```

Getting help

Like R, once a module or package has been imported, you can get information about how to run that package: the `help` command gives information about the package or function.

Type `"q"` to get out of help screen.

```
>>> import os
>>> help(os)
...
>>>
>>> import time
>>> help(time)
...
>>>
```

Making your own help

You can create your own help entry for your functions, called "docstrings".

- Put """ around your docstring text
- It must be at the start of your function.
- Use as many lines as you like.
- You can also make docstrings for the whole module.

```
>>> import listUtils
>>> help(listUtils.list2str)
Help on function list2str in module
listUtils:

list2str(lst)
    Function list2str converts every
    element of a list to a string value
```

```
# listUtils.py
"""
This module contains helper functions
"""


def list2str(lst):
    """
    Function list2str converts every
    element of a list to a string value
    """
    result = []
    for elem in lst:
        result.append(str(elem))
    return(result)
```

Command line arguments

We already saw how we can pass the parameters to the script using package `argparse`. Python has another way to retrieve these parameters

```
# cmd_param.py
import sys
print("There are", len(sys.argv), "arguments.")
print("The command line arguments are:", sys.argv)
```

```
$ python cmd_param.py param_1 99.9
There are 3 arguments.
The command line arguments are: ['cmd_param.py', 'param_1', '99.9']
```

Remember that by default, the arguments are strings.

List comprehensions

Very often we need to transform every element of a list. For example:

```
>>> mynums = [1, 2, 3, 4]
>>> squared_nums = []
>>> for num in mynums:
...     squared_nums.append(num * num)
...
>>> print(squared_nums)
[1, 4, 9, 16]
```

List comprehensions provide a concise way to create such lists.

```
>>> [num * num for num in mynums]
[1, 4, 9, 16]
>>> [num * num for num in mynums if num % 2 == 0]
[4, 16]
```

The % (modulo) operator calculates the remainder from the division.