Session 2: Python Tutorial - Flow Control

Prepared by:Ooi Yao Feng

1 Introduction

Flow control statements are where the rubber really meets the road in programming. Without it, a program is simply a list of statements that are sequentially executed. Flow control decides which Python instructions to execute under which conditions. The control flow of Python program is regulated by conditional statements, loops, and function calls. In this tutorial, we will cover the `if` statement, `for` and `while` loops.

1.1 Elements of Flow Control

Flow control statements often start with a part called the *condition* are always followed by a block of code called the *clause*.

1.1.1 Conditions

The Boolean expressions you've seen so far could all be considered conditions, which are the same thing as expressions; *condition* is just a more specific name in the context of flow control statements. Conditions always evaluate down to a Boolean value, `True` or `False`, and almost every flow control statement uses a condition.

1.1.2 A Brief Review on Logical Operators

Name	Operator	Meaning
Negation	not	True becomes False, and vice versa
Logical AND	x and y	True if both x and y are True, False otherwise
Logical OR	x or y	True if either x or y are True, False if both are False
Equality	x == y	True if x has the same value as y
Inequality	x != y	True if x does not have the same value as y
Comparative	x $\geq$ y	True if the value of x is greater or equal to y
...	...	...
Containment	x in y	True if x is an element/member of y
Identity	x is y	True if x and y 'point' to the same thing

Nota Belle: The equality operator and the identity operator are fundamentally different, though they can sometimes be used interchangeably in code. The identity operator only returns True when two variable names reference the same value stored in memory. Example: 1 is 1.0 returns False as 1 is int while 1.0 is a float data type. The identity operator can be faster than

"==" but it is far stricter, as in what is True in equality is not necessarily True in identity. Therefore, it is safer a lot of times to use the equality operator. The identity operator is safer to be used for 'singletons' like None.

1.1.3 Quick Reminder: Of blocks of codes and whitespaces

Lines of Python code can be grouped together in *blocks*. You can tell when a block begins and ends from the indentation of the lines of code. There are three rules for blocks.

- Blocks begins when the indentation increases
- Blocks can contain other blocks
- Blocks end when the indentation decreases to zero or to a containing block's indentation

Python detects block boundaries automatically, by line *indentation* - that is, the empty space to the left of your code. All statements indented the same distance to the right belong to the same block of code. In other words, the statements within a block line up vertically, as in a column. The block ends when the end of the file or a lesser-indented line is encountered, and more deeply nested blocks are simply indented further to the right than the statements in the enclosing block.

```
x = 1
if x:
    y = 2
    if y:
        print('block2')
    print('block1')
print('block0')
```

Listing 1: Indentation block.

2 `if` Statements

The most common type of flow control statement is the `if` statement. An `if` statement's clause will execute if the statement's condition is `True`. The clause is skipped if the condition is `False`. In other words, an `if` statement could be read as, "If this condition is true, execute the code in the clause." An `if` statement consists of the following:

- The `if` keyword
- A condition (an expression that evaluates to `True` or `False`)
- A colon
- Starting on the next line, an indented block of code (`if` clause)

Let's say you have some code that checks to see whether someone's name is James [1]:

[1] Note: This example is shown on Python interactive prompt.

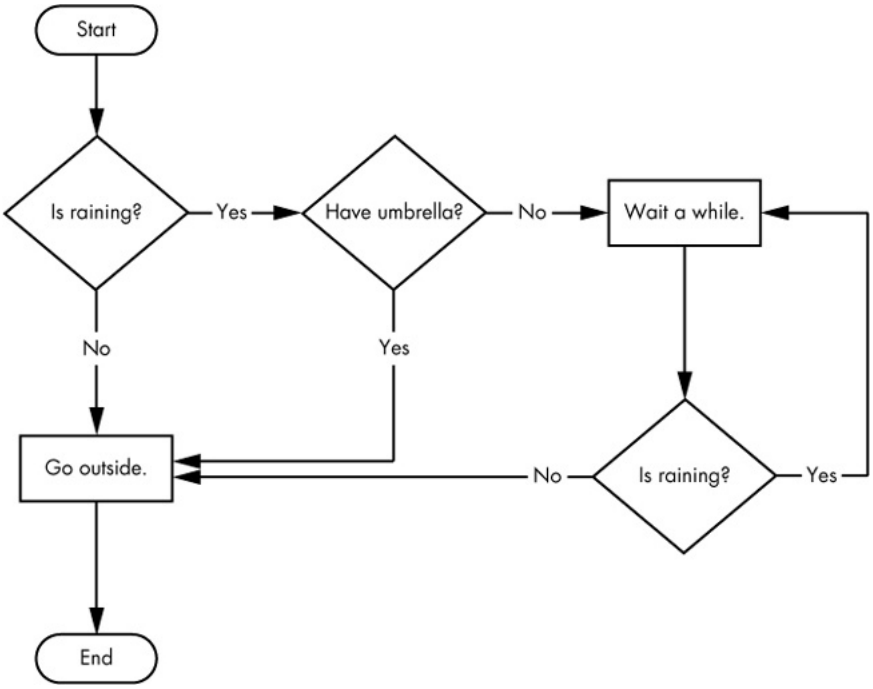

Figure 1: A flowchart to tell you what to do if it is raining.

```
>>> name = 'James'
>>> if name == 'James':
...     print('Hi, James.')
...
Hi, James.
```

2.1 `else` Statements

An `if` clause can optionally be followed by an `else` statement. The `else` clause is executed only when the `if` statement's condition is `False`. In plain English, an `else` statement could be read as, "If this condition is true, execute this code. Or else, execute that code." An `else` statement doesn't have a condition, and in code, an `else` statement always consists of the following:

- The `else` keyword
- A colon
- Starting on the next line, an indented block of code (`else` clause)

Returning to the James example, let us look at some code that uses an `else` statement to offer a different greeting if the person's name isn't James.

```
>>> name = 'Olive'
>>> if name == 'James':
...     print('Hi, James.')
... else:
...     print('Hello, stranger.')
...
Hello, stranger.
```

2.2 `elif` Statements

While only one of the `if` clauses will execute, you may have a case where you want one of *many* possible clauses to execute. The `elif` statement is an "else if" statement that always follows an `if` or another `elif` statement. It provides another condition that is checked only if all of the previous conditions were `False`. In code, an `elif` statement always consists of the following:

- The `elif` keyword
- A condition
- A colon
- Starting on the next line, an indented block of code (`elif` clause)

```
>>> x = 'killer rabbit'
>>> if x == 'roger':
...     print("shave and a haircut")
... elif x == 'bugs':
...     print("what's up doc?")
... else:
...     print('Run away! Run away!')
...
Run away! Run away!
```

This multiline statement extends from the `if` line through the block nested under the `else`. When it is run, Python executes the statements nested under the first test that is true, or the `else` part if all tests are false (in this example, they are). In practice, both the `elif` and `else` parts may be omitted, and there may be more than one statement nested in each section. Note that the words `if`, `elif` and `else` are associated by the fact that they line up vertically, with the same indentation.

2.3 General Form

```
if <condition1>:
    <if clause>
elif <condition2>:
    <elif clause>
else:
    <else clause>
```

Note: You can add more `elif` statements as many as you like.

2.4 Ternary Form

You can update/declare variables using ternary conditional operators such that you can compress the long if..else form before into a single statement. Note: The else operator must be included, otherwise it wouldn't work.

```
check1 = False
check2 = False

var1   = 6 if check1 else 5
print(var1)
```

```
var2   = 6 if check1 else (5 if check2 else 4)
print(var2)
```

2.5 More Examples

```
a = 200
b = 33
c = 500
if a > b or a > c:
    print("At least one of the conditions is True")
```

```
isTall = True
isMale = False
if isTall and isMale:
    print("You are a tall male.")
elif not isTall and isMale:
    print("You are a short male.")
elif isTall and not isMale:
    print("You are a tall female.")
else:
    print("You are a short female.")
```

3 Exceptions

Sometimes you want to add some new lines into your code which was functioning well, but constant bugs are usually annoying. You can use the try...except operators to literally try out a line of code to see if it is working fine or not. This helps to mitigate catastrophic crashes when running.

```
try:
    <try clause>
except:
    <except clause>
```

```
num1 = 0
num2 = 1
try:
    print(num2/num1)
except:
    print("You can't divide by zero!!!!!)
```

4 `while` Loops

Python's `while` statement is the most general iteration construct in the language. In simple terms, it repeatedly executes a block of statements as long as a test at the top keeps evaluating to a true value. It is called a "loop" because control keeps looping back to the start of the statement until the test becomes false. When the test becomes false, control passes to the statement that follows the `while` block. In code, a `while` statement always consists of the following:

- The `while` keyword
- A condition
- A colon
- Starting on the next line, an indented block of code (`while` clause)

You can see that a `while` statement looks similar to an `if` statement. The difference is in how they behave. At the end of an `if` clause, the program execution continues after the `if` statement. But at the end of a `while` clause, the program execution jumps back to the start of the `while` statement.

4.1 Examples

Let us look into some simple examples. The first, which consists of a `print` statement nested in a `while` loop, just print a message forever. Python keeps executing the body forever, or until you stop its execution. This sort of behavior is usually called an *infinite loop* - it is not really immortal, but you may need a **Ctrl-C** key combination to forcibly terminate one.

```
>>> while True:
...     print('Type Ctrl-C to stop me!')
```

The next example keeps slicing off the first character of a string until the string is empty and hence false.

```
>>> x = 'spam'
>>> while x:
...     print(x, end=' ')
...     x = x[1:]
...
spam pam am m
```

The `end=' '` keyword argument used here to place all outputs on the same line separated by space. The following code counts from the value of `a` up to, but not including `b`.

```
>>> a=0; b=10
>>> while a < b:
...     print(a, end=' ')
...     a += 1
...
0 1 2 3 4 5 6 7 8 9
```

4.2 Exercise

Using the Fibonacci Sequence and the while loop, print out its numbers and terminate the loop as it reaches 100. Hint:

$$F_n = F_{n-1} + F_{n-2} \tag{1}$$

5 `break`, `continue` and `pass` Statements

5.1 `break`

There is a shortcut to get the program execution to break out of a `while` loop's clause early. If the execution reached a `break` statement, it immediately exits the loop's clause. In code, a `break` statement simply contains the `break` keyword.

```
while True:
    name = input('Please type your name"')
    if name == 'your name':
        break
print('Thank you!')
```

The program asks the user to enter `your name`. While the execution is still inside the `while` loop, an `if` statement checks whether `name` is equal to `'your name'`. If this condition is `True`, the `break` statement is run, and the execution moves out of the loop to `print('Thank you!')`. Otherwise, the `if` statement's clause that contains the `break` statement is skipped, which puts the execution at the end of the `while` loop. At this point, the program execution jumps back to the start of the `while` statement to recheck the condition.

5.2 `continue`

Like `break` statements, `continue` statements are used inside loops. When the program execution reached a `continue` statement, the program execution immediately jumps back to the start of the loop and reevaluates the loop's condition.

```
while True:
    name = input('Who are you?')
    if name != 'Joe':
        continue
    password = input('Hello, Joe. What is the password? (It is a fish.)')
    if password == 'swordfish':
        break
print('Access granted')
```

If the user enters any name besides `Joe`, the `continue` statement causes the program execution to jump back to the start of the loop. When the program reevaluates the condition, the execution will always enter the loop, since the condition is simply the value `True`. Once the user makes it past that `if` statement, they are asked for a password. If the password entered is `swordfish`, then the `break` statement is run, and the execution jumps out of the `while` loop to print `Access granted`. Otherwise, the execution continues to the end of the `while` loop, where it then jumps back to the start of the loop.

```
i = 1
while i < 6:
    print(i)
    if i == 3:
        break
    i += 1
```

```
i = 1
while i < 6:
    print(i)
```

```
    if i == 3:
        continue
    i += 1
```

```
i = 1
while i < 6:
    print(i)
    i += 1
else:
    print("i is no longer less than 6")
```

5.3 pass

In Python 3.x, the `pass` statement is essentially a null statement. It is not ignored by Python nor does it do anything. The only reason to use it is if you want a placeholder statement for your code. For example, you are building a loop or a function (which you will learn in the next sessions) but you do not want to fully implement it just yet.

```
i = 0
while i<100:
    if i%2 == 0:
        pass
    else:
        print("This number is not a multiplicative of two.")
    i+=1
```

6 `for` Loops

The `for` loop is a generic iterator in Python: it can step through the items in any ordered sequence or other iterable object. The `for` statement works on strings, lists, tuples, and other built-in iterables.

6.1 General Form

The Python `for` loop begins with a header line that specifies an assignment target (or targets), along with the object you want to step through. The header is followed by a block of statements that you want to repeat.

```
for <target> in <object>:
    <clause>
```

6.2 Examples

In our first example, we will assign the name `x` to each of the three items in a list in turn, from left to right, and the `print` statement will be executed for each. Inside the `print` statement (the loop body), the name `x` refers to the current item in the list.

```
>>> for x in ["spam", "eggs", "ham"]:
...     print(x, end=' ')
...
spam eggs ham
```

The next two examples compute the sum and product of all the items in a list.

```
>>> sum = 0
>>> for x in [1, 2, 3, 4]:
...     sum = sum + x
...
>>> sum
10
>>> prod = 1
>>> for item in [1, 2, 3, 4]: prod *= item
...
>>> prod
24
```

6.3 `range()` Function

To loop through a set of code a specified number of times, we can use the `range()` function. The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

```
>>> for x in range(6):
...     print(x)
...
0
1
2
3
4
5
>>>
>>> for i in range(0, 10, 2):
...     print(i)
...
0
2
4
6
8
```

Points to remember about Python `range()` function arguments

- `range()` function only works with the integers. All arguments must be integers
- All three arguments can be positive or negative
- The step value must not be zero. If a step is zero, python raises a `ValueError` exception

A strange thing happens if you just print a range.

```
>>> print(range(10))
range(0, 10)
```

From Python documentation: *In many ways the object returned by* `range()` *behaves as if it is a list, but in fact it isn't. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn't really make the list, thus saving space.* Thus, maybe you are curious about how to get a list from a range. Here is the solution:

```
>>> print(*range(6))
0 1 2 3 4 5
```

```
>>> list(range(4))
[0, 1, 2, 3]
```

7 Exercises

Task 1. Write a program to check if a year is leap year or not.
Task 2. Write a program to ask the user for a number. Print out which category the number is in: `positive`, `negative` or `zero`.
Task 3. Write a program that counts the number of elements within a list that are greater than 30.
Task 4. Write a Python program to construct the following pattern:

```
1
22
333
4444
55555
666666
7777777
88888888
999999999
```

Task 5. Given the 3 sides of a triangle (x, y and z), determine whether the triangle is equilateral, isosceles or obtuse. (Also include a check at the beginning to see whether the lengths of the sides satisfy the triangle inequality.)