Scientific Computing 272

Section 6: Repetition in Python

Last updated: 22 July 2019

UNIVERSITEIT STELLENBOSCH UNIVERSITY

Willem Bester

Section Outline

Counted loops

`while` loops

File input

Controlling loops

Revision

Conditional statements let us decide whether or not to something, and loops let us do things many times. We have seen the `for` loop:

```
for ⟨variable⟩ in ⟨list⟩:
    ⟨block⟩
```

It is useful for doing something with each value. But what if we want to change the list?

Example

```
>>> values = [2, 3, 5, 7]
>>> for v in values:
...     v = 2 * v
...
>>> values
[2, 3, 5, 7]
```

To change a list, we need to repeat statements with indices like `values[0] = 2 * values[0]`. [Why?]

Ranges of numbers

- The built-in function `range` generates a list of numbers
- It takes one, two, or three parameters:
  - `range(`n`)` generates a list of the integers in $[0, n)$
  - `range(`m`,` n`)` generates a list of the integers in $[m, n)$
  - `range(`m`,` n`,` s`)` generates a list of the integers in $[m, n)$, where $s \neq 0$ is the **step size**
- It is deliberately consistent with how sequence indexing work: It works from m up to, but not including, n
- `range(`n`)` is equivalent to `range(`0`,` n`)`
- `range(`m`,` n`)` is equivalent to `range(`m`,` n`,` 1`)`
- If $s > 0$, we must give $m < n$, else the list is empty
- If $s < 0$, we must give $m > n$, else the list is empty
- If $s = 0$ … well, don't do it, the computer might blow up

Example

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(2, 7)
[2, 3, 4, 5, 6]
>>> range(-3, 4)
[-3, -2, -1, 0, 1, 2, 3]
>>> range(-3, 4, 2)
[-3, -1, 1, 3]
>>> range(-3, 4, -2)
[]
>>> range(2030, 2000, -4)
[2030, 2026, 2022, 2018, 2014, 2010, 2006, 2002]
>>> int_sum = 0
>>> for i in range(1, 101):
...     int_sum += i
...
>>> int_sum   # sum of integers from 1 to 100
5050
```

Ranges of numbers

- With the `range` function, we can use a `for` loop to iterate over the indices of a list
- Use the function `len` to get the number of elements in a list
- If `list` is a list, `range(len(list))` gives the list of valid list indices

Example

Suppose that we want to double the value of each element in a list.

```
>>> list = [2, 3, 5, 7, 11, 13, 17, 19]
>>> for i in range(len(list)):
...     list[i] *= 2
...
>>> list
[4, 6, 10, 14, 22, 26, 34, 38]
>>>
```

- Bottom line: If you want to change a list element, access it by index

Enumerating over a list

- Given a sequence—a list, a tuple, or a string—the function `enumerate` returns a list of pairs:
  - The first element is the index
  - The second element is the value at this index in the sequence
- Note that this pair is a two-element tuple, and hence, cannot be mutated

Example

```
>>> for x in enumerate('abc'):
...     print(x)
...
(0, 'a')
(1, 'b')
(2, 'c')
>>>
```

Enumerating over a list

Example

```
>>> values = [37, 41, 43]
>>> for pair in enumerate(values):
...     i = pair[0]
...     v = pair[1]
...     values[i] = 2 * v
...
>>> values
[74, 82, 86]
>>> print(i, v)
2 43
```

- After the `for`, `i` and `v` have the the values they were assigned in the last iteration
- we must still access the list by index to change it

Multivalued assignment

- Python allows **multivalued assignment**
- If there are more than one variable on the left side of an assignment and an equal number of values on the right:
  - Python matches them up
  - And does all the assignments at once

Example

```
>>> x, y = 1, 2
>>> print('x =', x, '; y =', y)
x = 1 ; y = 2
>>> x, y = y, x
>>> print('x =', x, '; y =', y)
x = 2 ; y = 1
```

Multivalued assignment

- Multivalued assignment also works if the values on the right are in a sequence
- Python "explodes" the sequence on the right and then assigns them to the variables on the left

Example

```
>>> first, second, third = [1, 2, 3]
>>> print(', , '.format(first, second, third))
1, 2, 3
>>> first, second, third = 'abc'
>>> print(', , '.format(first, second, third))
a, b, c
```

Multivalued assignment

Example

```
>>> list = [37, 41, 43]
>>> for (i, v) in enumerate(list):
...     list[i] = 2 * v
...
>>> list
[74, 82, 86]
>>> print(i, v)
2 43
```

- For each iteration, `enumerate` produces a tuple
- Using `(i, v)` after `for` makes Python break the tuple apart, assigning the first element to `i` and the second to `v`
- After the `for`, `i` and `v` have the the values they were assigned in the last iteration

Ragged lists

Nested lists may have unequal lengths. For non-uniform data they may be tricky to process; however, they do arise naturally in some contexts.

Example

Say, for example, for a couple of days I log the times my power goes out. (Okay, in reality we would need a supercomputer to model Eskom's downtimes.)

```
>>> times = [["9:02", "10:17", "13:52" "21:15"],
... ["8:45", "13:44", "14:13"],
... ["8:55", "11:11", "12:34", "18:23", "21:31"]]
>>> for day in times:
...     for time in day:
...         print(time, end=" ")
...     print()
...
9:02 10:17 13:52 21:15
8:45 13:44 14:13
8:55 11:11 12:34 18:23 21:31
```

`while` loops

- If you know the number of times a loop is to execute a `for` loop suffices
- Sometimes it is impossible to know, and then we use `while`:

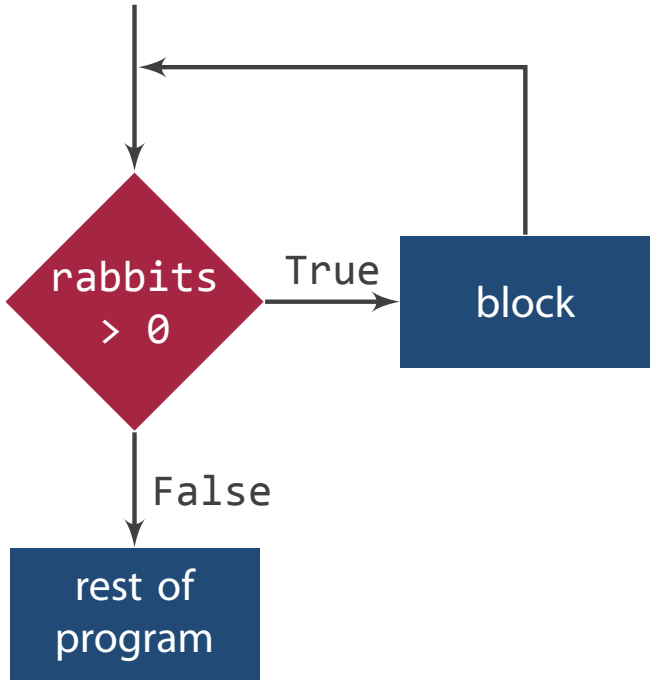
```
while ⟨condition⟩:
    ⟨block⟩
```

- The ⟨**condition**⟩ is a boolean expression just like for an `if`
- When Python encounters a `while`
  1. It evaluates ⟨**condition**⟩
  2. If the ⟨**condition**⟩ is false, it skips the ⟨**block**⟩
  3. If the ⟨**condition**⟩ is true, it executes the ⟨**block**⟩ and jumps back to item 1
- So, a `while` is executed until the ⟨**condition**⟩ is false
- If the ⟨**condition**⟩ is false to start with, the ⟨**block**⟩ is not executed at all

`while` loops

Example

```
>>> rabbits = 3
>>> while rabbits > 0:
...     print(rabbits)
...     rabbits -= 1
...
3
2
1
```



- Note that the loop did not print 0
- When the number of rabbits reaches 0, the `while` condition is false

Example (`population.py`)

Suppose that we calculate the growth of a bacterial colony with an exponential model

$$P(t+1) = P(t) + rP(t),$$

where $P(t)$ is the population at time t, and r is the growth rate. We want to know how long it takes the bacteria to double their numbers.

```
t = 0       # minutes
pop = 1000 # bacteria to start with
r = 0.21   # 21% growth per minute
while pop < 2000:
    pop += pop * r
    print(pop)
    t += 1
print("{} min. for bacteria to double".format(t))
print("Final population is {:6.2f}".format(pop))
```

Example

Example (output)

```
1210.0
1464.1
1771.561
2143.5888099999997
4 min. for bacteria to double
Final population is 2143.59
```

- Because the time variable *t* was inside the loop, its value after the loop is the time of the last iteration
- This the time we want, since the colony became (more than double) its original size during the last iteration
- Can we write the loop condition in another way?

Input from files

- Use the Python function `open` to access a file
- The first parameter names the file
- The second parameter indicates the access mode:
  - `'r'` for reading
  - `'a'` for appending (to end of existing data in file)
  - `'w'` for writing (erase everything, and start from scratch)
- The result returned **is not** the contents of the file, but a **file object**, whose methods allow access to the contents of the file
- The method `read` allows access to the individual bytes in the file
- However, for data processing, we often work with text data
- `readline` reads the next line of text from the file
- A line is all the characters up to and including the next end-of-line marker
- An empty line is returned when no more data is available

Example

We have a text file `planets.txt` with the following contents:

```
Mercury
Venus
Earth
Mars
```

Do the following in the Python interpreter:

```
>>> file = open('planets.txt', 'r')
>>> for line in file:
...     print(line.rstrip(), len(line))
...
Mercury 8
Venus 6
Earth 6
Mars 5
```

- Python automatically calls `readline` of the file in a `for` loop
- The end-of-line character is included in the line length

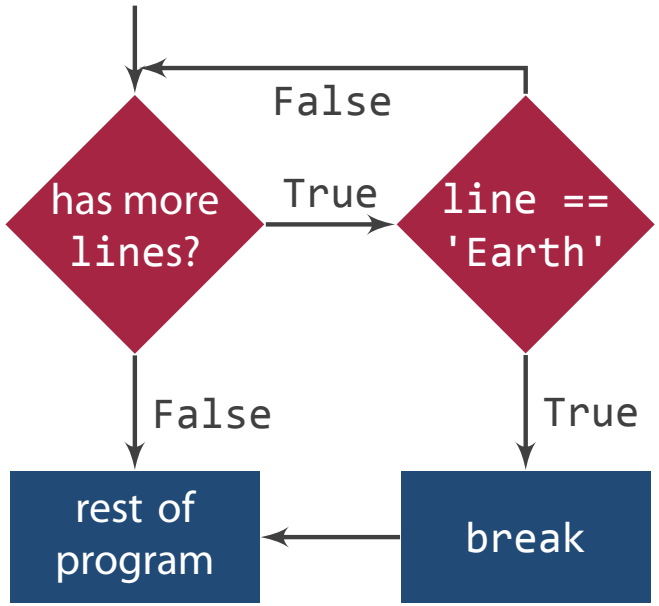
Controlling loops

- As a rule, `for` and `while` loops execute all the body statements on each iteration
- Sometimes it is useful to break this rule
- The `break` statement exits the loop body immediately, and execution resumes with the first statement after the loop
- Note: `break` only exits the innermost loop that contains it
- In a nested loop, `break` in the inner loop will only exit the inner loop, not both loops
- The `continue` statement immediately starts the next iteration of the loop and skips any statements in the loop body that appear after it
- It is possible to get by without these statements
- However, their use may result in clearer code with fewer levels of indentation

The break statement

Example

```
earth_line = 1
file = open("planets.txt", "r")
for line in file:
    line = line.strip()
    if line == "Earth":
        break
    earth_line += 1
print("Earth at line", earth_line)
```



- The `for` loop terminates as soon as it gets to the first line that is the string “Earth”
- Because the end-of-line character is included in a line read from a file, we first have to strip it

The `continue` statement

- Comments are quite useful, also in our own data files
- We can use `continue` to skip comments in our files, similar to what happens in Python files

Example

Say that we a file with the planets ordered by weight.

```
earth_line = 1
file = open("planets.txt", "r")
for line in file:
    line = line.strip()
    if line.startswith("#"):
        continue
    if line == "Earth":
        break
    earth_line += 1
print("Earth is {}th-lightest".format(earth_line))
```