Iterations and loops in Python

Petr Pošík

Department of Cybernetics, FEE CTU in Prague

EECS, BE5B33PRG: Programming Essentials, 2015

Prerequisities:

- strings, tuples, lists
- functions

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

Repeated execution of a set of statements is called **iteration**. Because iteration is so common, Python provides several language features to make it easier.

Loop types

Many computer languages have at least 2 types of loops:

- **`for`-loops** suitable when a piece of code shall be ran a number of times with a known number of repetitions, and
- **`while`-loops** when the number of iterations is not known in advance.

`For`-loops

The general form of a for statement is as follows:

```
for <variable> in <sequence>:
    <block of code>
```

Execution:

- The `<variable>` following `for` is bound to the first value in the `<sequence>`, and the `<code block>` is executed.
- The `<variable>` is then assigned the second value in the `<sequence>`, and the `<code block>` is executed again.
- ...

The process continues until the sequence is exhausted, or a `break` statement is executed within the code block.

For or For-each?

A good way of thinking about the `for`-loop is to actually read it as `for each` like:

```
for each item in a set (or in a sequence, or in a collection):
    do something with the item
```

Iterating over characters of a string

The for-loop can be easilly used to iterate over characters of a string.

Write a program that computes the sum of all digits in a given string:

In [1]:
```
string = '123456789'
```

As a piece of code:

In [2]:
```
sum = 0
for char in string:
    sum += int(char)
print(sum)
```

```
45
```

As a function:

In [3]:
```
# Compute the sum of the numbers in the string
def sum_num(s):
    """Sum the numerals in a string"""
    sum = 0
    for ch in string:
        sum = sum + int(ch)
    return sum

print(sum_num(string))
```

```
45
```

Iterating over tuples and lists

Actually the same as above:

In [4]:
```
for i in [1, 2, 3]:
    print(2*i, end=', ')
```

```
2, 4, 6,
```

In [5]:
```
for word in ['Hello!', 'Ciao!', 'Hi!']:
    print(word.upper(), end=', ')
```

```
HELLO!, CIAO!, HI!,
```

The `range` function

To reiterate, the `range` function can be used to generate a sequence of numbers:

In [6]:
```
for i in range(1,4):
    print(2*i, end=", ")
```

```
2, 4, 6,
```

The `range` function is a generator, i.e. it does not produce the whole sequence when called; rather, it generates the next number of the sequence when asked. To collect all the numbers of the sequence, we need to enclose the call to `range` in a `list`.

In [7]:
```
print(list(range(10)))
start, end, step = 13, 2, -3
print(list(range(start, end, step)))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[13, 10, 7, 4]
```

Nested `for`-loops

When a for-loop is inside the code block of another for-loop, we call them **nested**. Two nested loops can be used e.g. to generate all pairs of values from two sequences:

In [8]:
```
# Print out part of the multiplication table
# Your code here!
```

In [9]:
```
for i in range(1,4):
    for j in range(1,6):
        print(i*j, end=", ")
    print()
```

```
1, 2, 3, 4, 5,
2, 4, 6, 8, 10,
3, 6, 9, 12, 15,
```

The `enumerate` function

Sometimes we need to go not only through the items of a sequence, but also need their indices. Let's have a list of work days like this:

In [10]:
```
workdays = 'Monday Tuesday Wednesday Thursday Friday'.split()
print(workdays)
```

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

Suppose we want to print out the work days and their number. The classic way to do this would be:

In [11]:
```
for i in range(len(workdays)):
    print('The day nr.', i, 'is', workdays[i])
```

```
The day nr. 0 is Monday
The day nr. 1 is Tuesday
The day nr. 2 is Wednesday
The day nr. 3 is Thursday
The day nr. 4 is Friday
```

Python has a function (generator) `enumerate`. It produces pairs of item index and the index itself:

In [12]:
```
list(enumerate(workdays))
```

Out[12]:
```
[(0, 'Monday'),
 (1, 'Tuesday'),
 (2, 'Wednesday'),
 (3, 'Thursday'),
 (4, 'Friday')]
```

We can use it in the for-loop as follows.

In [13]:
```
for i, day_name in enumerate(workdays):
    print('The day nr.', i, 'is', day_name)
```

```
The day nr. 0 is Monday
The day nr. 1 is Tuesday
The day nr. 2 is Wednesday
The day nr. 3 is Thursday
The day nr. 4 is Friday
```

Note that after the `for` command we have a pair of variables instead of the usual single variable. (This can be even generalized and we can easily iterate over n-tuples.) We do not have to measure the length of the list, we do not have to index into the list. This approach is more Pythonic than the previous one.

Iterating over 2 sequences at once

How to compute scalar product of 2 vectors?

In [14]:
```
def scalar_product(v1, v2):
    # Your code here
    pass
```

In [15]:
```
vec1 = [1,2,3,4]
vec2 = [2,2,1,1]
print(scalar_product(vec1, vec2))
```

```
None
```

In [16]:
```
def scalar_product(v1, v2):
    sp = 0
    for i in range(len(v1)):
        sp += v1[i] * v2[i]
    return sp
```

The `zip` function

Function `zip()` takes n iterables (sequences) and creates a single sequence consisting of n-tuples formed by the first, second, ... items of the sequences.

In [17]:
```
s1 = ['a','b','c','d']
s2 = [1,2,3,4]
list(zip(s1, s2))
```

Out[17]:
```
[('a', 1), ('b', 2), ('c', 3), ('d', 4)]
```

Scalar product with the help of zip function:

In [18]:
```
def scalar_product_zip(v1, v2):
    sp = 0
    for x, y in zip(v1, v2):
        sp += x*y
    return sp
print(scalar_product_zip(vec1, vec2))
```

```
13
```

Iterating over nested lists

Suppose you have a list of lists (or similar nested data structure), where the inner lists may be of different lengths. You want to go over all items in the nested lists.

In [19]:
```
elements = [['Li', 'Na', 'K'], ['F', 'Cl', 'Br', 'I']]
for element_group in elements:
    print(element_group)
```

```
['Li', 'Na', 'K']
['F', 'Cl', 'Br', 'I']
```

In [20]:
```
for element_group in elements:
    for element in element_group:
        print(element, end=' ')
```

```
Li Na K F Cl Br I
```

The `break` and `continue` statements

The `break` and `continue` statements change the flow of control in for- and while-loops.

The `break` statement

When the **`break`** command is issued inside a loop, it ends the innermost loop prematurely, passing control to the code right after the loop.

Often it may be convenient to construct an infinite while-loop by starting it with `while True:`, and ending the cycle by calling `break` when some condition is satisfied inside the loop.

The `continue` statement

When the **`continue`** command is issued inside a loop, the current iteration is interrupted by skipping the rest of the loop code block, and transfering control to the begining of the next iteration, i.e.

- in case of for-loop, the control variable is bound to the next item of the sequence,
- in case of while-loop, the test is re-evaluated, and the next iteration is started.

Very often, it is convenient to use `continue` to filter out the iterations for which the loop should not be executed.

Examples: `break` and `continue`

Using `continue` to filter out iterations for numbers divisible by 2 or 3:

In [21]:
```
for i in range(15):
    if i % 2 == 0 or i % 3 == 0:
        continue
    print(i, end=',')
```

```
1,5,7,11,13,
```

Using `continue` to select the uppercase letters only:

In [22]:
```
thing = "Something Posing As Meat"
for char in thing:
    if not char.isupper(): continue
    print(char, end='')
```

```
SPAM
```

Using `break` to find the first occurance of a char in a string:

In [23]:
```
string = 'Bananarama'
char_to_find = 'r'
pos = None
for i, c in enumerate(string):
    if c == char_to_find:
        pos = i
        break
print(pos)
```

```
6
```

Using `break` statement to exhaustively search for a cube root of a non-negative integer:

In [24]:
```
def cube_root(number):
    """Exhaustively search for the integer cube root of non-negative integer."""
    for guess in range(number+1):
        if guess**3 >= number: break
    if guess**3 == number:
        return guess
    else:
        return None
```

In [25]:
```
cube_root(27)
```

Out[25]:
```
3
```

In [26]:
```
print(cube_root(28))
```

```
None
```

The `while` statement

While loop represents a generic **iteration** mechanism.

```
while <Boolean expression>:
    <block of code>
```

Execution:

- Perform a test (evaluate the `<Boolean expression>`).
- If the test evaluates to `True`, executes the loop body (`<block of code>`) once, and return back to reevaluate the test.
- Repeat this process until the test evaluates to `False`; then pass the control to the code following the `while` statement.

Example: Square of an integer

Code which computes a square of an integer `n` by `n`-times adding the value of `n`:

In [27]:
```
def square_int(n):
    """Return the square of int n by n times adding n."""
    ans = 0
    iters_left = n
    while iters_left != 0:
        ans = ans + n
        iters_left = iters_left - 1
    return ans
```

In [28]:
```
n = 10
print(n, '*', n, '=', square_int(n))
```

```
10 * 10 = 100
```

Let's redefine the function by allowing it to print some info during the iterations.

In [29]:
```
def square_int(n, info=False):
    """Return the square of int n by n times adding n."""
    ans = 0
    iters_left = n
    if info:
        print('Iterations left:', iters_left, '| Current value of ans:', ans)
    while iters_left != 0:
        ans = ans + n
        iters_left = iters_left - 1
        if info:
            print('Iterations left:', iters_left, '| Current value of ans:', ans)
    return ans
```

In [30]:
```
n = 6
print(n, '*', n, '=', square_int(n, info=True))
```

```
Iterations left: 6 | Current value of ans: 0
Iterations left: 5 | Current value of ans: 6
Iterations left: 4 | Current value of ans: 12
Iterations left: 3 | Current value of ans: 18
Iterations left: 2 | Current value of ans: 24
Iterations left: 1 | Current value of ans: 30
Iterations left: 0 | Current value of ans: 36
6 * 6 = 36
```

Example: Growth of a bacterial colony

We can calculate the growth of a bacterial colony using simple exponential growth model:

$$P(t+1) = P(t) + r \cdot P(t) = (1+r)P(t),$$

where $P(t)$ is a population size at time t, and r is the growth rate.

Let's see how long it takes the bacteria to double their numbers for $r = 0.21$:

In [31]:
```
time = 0
ini_popsize = 1000  # 1000 bacteria in the beginning
growth_rate = 0.21  # 21 % growth per minute
```

In [32]:
```
# Your code here
```

In [33]:
```
popsize = ini_popsize
while popsize < 2 * ini_popsize:
    popsize = popsize + growth_rate * popsize
    time += 1
    print('After', time, 'minutes, the population size is', round(popsize))
```

```
After 1 minutes, the population size is 1210
After 2 minutes, the population size is 1464
After 3 minutes, the population size is 1772
After 4 minutes, the population size is 2144
```

Examples

Square root of any positive number

Implement an algorithm that can compute the square root of any positive number. How can we do that?

Options:

- by exhaustive enumeration of discretized space,
- by bisection search, and
- by the Newton-Raphson method.

Discretization + exhaustive search

Do you know how to do that? Do you know everything you need to know?

Needs better specs. $\sqrt{2}$ is not rational number, there is no way to represent the result precisely. So the problem cannot be solved.

We should find a reasonably precise **approximation** to the square root, i.e. we should find a number which is close enough - its square is in a distance at most `epsilon` from the given square.

The following code approximates square root by exhaustive enumeration.

In [34]:
```
def sqrt_exhaustive(x, epsilon=0.01):
    # Your code here
    pass
```

In [35]:
```
def sqrt_exhaustive(x, epsilon=0.01):
    """Compute square root of positive number by exhaustive enumeration."""
    step = epsilon**2 # Or any other value, like 0.1, 0.0001, etc.
    n_guesses = 0
    guess = 0.0
    while abs(guess**2 - x) >= epsilon and guess**2 <= x:
        guess += step
        n_guesses += 1
    if abs(guess**2 - x) >= epsilon:
        return None, n_guesses
    return guess, n_guesses
```

In [36]:
```
sqrt_exhaustive(0.25)
```

Out[36]:
```
(0.48989999999996237, 4899)
```

The algorithm will make about `sqrt(x) / step` steps.

What if we try a bit larger number?

In [37]:
```
sqrt_exhaustive(123456)
```

Out[37]:
```
(None, 3513631)
```

Why didn't we find a solution? There surely is a floating point number that approximates the square root of 123456 to within 0.01. The program did not find it because the step was too large and it jumped over the solution. Try to make `step` equal to `epsilon**3`. It will eventually find the answer, but you might not have the patience to wait for it.

Bisection

Suppose we know that a good approximation to the square root of `x` lies somewhere between `0` (`low`) and `n` (`high`). We exploit the fact that the numbers are **ordered**. We can start by guessing that the approximation lies in the middle between `low` and `high`. If the guess is not correct, it can be either too low, so the solution must lie in the right half, or it is too high, then the solution must be in the left half. And we can apply the same method again in a smaller interval.

In [38]:
```
def sqrt_bisection(x, epsilon=0.01, info=False):
    """Compute square root of positive number by bisection."""
    # Your code here
    pass
```

In [39]:
```
def sqrt_bisection(x, epsilon=0.01, info=False):
    """Compute square root of positive number by bisection."""
    n_guesses = 0
    low = 0.0
    high = max(1.0, x)
    guess = (high + low) / 2
    while abs(guess**2 - x) >= epsilon:
        if info:
            print('low =', low, ' guess =', guess, ' high =', high)
        n_guesses += 1
        if guess**2 < x:
            low = guess
        else:
            high = guess
        guess = (high + low) / 2
    return guess, n_guesses
```

In [40]:
```
sqrt_bisection(0.0025)
```

Out[40]:
```
(0.0625, 3)
```

In [41]:
```
sqrt_bisection(25, info=True)
```

```
low = 0.0  guess = 12.5  high = 25
low = 0.0  guess = 6.25  high = 12.5
low = 0.0  guess = 3.125  high = 6.25
low = 3.125  guess = 4.6875  high = 6.25
low = 4.6875  guess = 5.46875  high = 6.25
low = 4.6875  guess = 5.078125  high = 5.46875
low = 4.6875  guess = 4.8828125  high = 5.078125
low = 4.8828125  guess = 4.98046875  high = 5.078125
low = 4.98046875  guess = 5.029296875  high = 5.078125
low = 4.98046875  guess = 5.0048828125  high = 5.029296875
low = 4.98046875  guess = 4.99267578125  high = 5.0048828125
low = 4.99267578125  guess = 4.998779296875  high = 5.0048828125
low = 4.998779296875  guess = 5.0018310546875  high = 5.0048828125
```

Out[41]:
```
(5.00030517578125, 13)
```

In [42]:
```
sqrt_bisection(123456)
```

Out[42]:
```
(351.36305049061775, 30)
```

Notice several things:

- It found a different approximation than the previous function. Both solutions meet the problem specifications.
- It needed only 13 steps to find the answer to $\sqrt{25}$. When you try to find $\sqrt{123456}$, only 45 iterations are needed.
- The search space is cut in half each iteration. When doing enumeration (previous case), we reduce the search space by a small amount only each iteration.
- There is nothing special about using this algorithm to find the square roots. With a small modifications you can use it to compute cube roots. Generally, you can find a root of any function, once you find a `low` and `high` limit bracketing the root.

Newton-Raphson

The Newton-Raphson algorithm is the most often used approximation algorithm. It can find real roots of many continuous functions, but here we concentrate on polynomials. The problem of finding an approximation g for the square root of 24 can be formulated as finding a g such that $f(g) = g^2 - 24 = 0$. Square root of a general number x is approximated by finding root of $f(g) = g^2 - x = 0$.

Newton proved a theorem that implies that if a value, call it g as *guess*, is an approximation to a root of $f(g)$, the

$$g - \frac{f(g)}{f'(g)},$$

where f' is the first derivative of f, is a better approximation.

For any constant b and any coefficient a, the first derivative of $ax^2 + b$ is $2ax$. Thus, to improve our guess g of square root of x, we can use

$$g_{new} \leftarrow g - \frac{f(g)}{f'(g)} = g - \frac{g^2 - x}{2g} = \frac{g + \frac{x}{g}}{2}.$$

In [43]:
```
def sqrt_NR(x, epsilon=0.01, info=False):
    # Your code here
    pass
```

In [44]:
```
def sqrt_NR(x, epsilon=0.01, info=False):
    """Compute square root of a positive number using Newton-Raphson."""
    guess = x/2
    n_guesses = 1
    if info:
        print('guess =', guess)
    while abs(guess**2 - x) >= epsilon:
        guess = guess - (guess**2 - x) / (2 * guess)
        n_guesses += 1
        if info:
            print('guess =', guess)
    return guess, n_guesses
```

In [45]:
```
sqrt_NR(25, info=True)
```

```
guess = 12.5
guess = 7.25
guess = 5.349137931034482
guess = 5.011394106532552
guess = 5.000012953048684
```

Out[45]:
```
(5.000012953048684, 5)
```

In [46]:
```
sqrt_NR(123456)
```

Out[46]:
```
(351.3630601485824, 12)
```

Note that now we needed only 5 guesses to find $\sqrt{25}$ (compared to 13 guesses using bisection), and only 12 guesses to find $\sqrt{123456}$ (compared to 45 using bisection). Still a considerable improvement.

Notebook config

Some setup follows. Ignore it.

In [47]:
```
from notebook.services.config import ConfigManager
cm = ConfigManager()
cm.update('livereveal', {
              'theme': 'Simple',
              'transition': 'slide',
              'start_slideshow_at': 'selected',
              'width': 1268,
              'height': 768,
              'minScale': 1.0
})
```

Out[47]:
```
{'height': 768,
 'minScale': 1.0,
 'start_slideshow_at': 'selected',
 'theme': 'Simple',
 'transition': 'slide',
 'width': 1268}
```

In [1]:
```
%%HTML
<style>
.reveal #notebook-container { width: 90% !important; }
.CodeMirror { max-width: 100% !important; }
pre, code, .CodeMirror-code, .reveal pre, .reveal code {
    font-family: "Consolas", "Source Code Pro", "Courier New", Courier, monospace;
}
pre, code, .CodeMirror-code {
    font-size: inherit !important;
}
.reveal .code_cell {
    font-size: 130% !important;
    line-height: 130% !important;
}
</style>
```