Chapter 4

Iteration over values

Programs often need to do repetitive things. Consider this example where \( x \) is assigned a value that is then printed:

\[
\begin{align*}
  x &= 1 \\
  \text{print}(x) \\
  x &= 5 \\
  \text{print}(x) \\
  x &= 3 \\
  \text{print}(x) \\
  x &= 7 \\
  \text{print}(x)
\end{align*}
\]

You can see that we do the same thing four times with the only difference that \( x \) takes a new value each time. Now carefully write this alternative code and compare what is printed to what was printed in the above example.

\[
\text{for } x \text{ in } [1, 5, 3, 7]: \\
  \text{print}(x)
\]

It should be exactly the same. What you just wrote is called a for-loop. It is called a for-loop because it does something for each of many values - in this case for each value in our list.

The statements nested under the for-loop are run as many times as there are values in our list, and every time they are run \( x \) has a new value. The first time the statements are run, \( x \) is assigned the first value in the list. The second time they are run \( x \) is assigned the second value in the list. This continues until \( x \) has been assigned all the values in the list.

The semantics of a for-loop are as follows:

1. First you write `for`.
2. Then you write the name of the variable that will be assigned a new value each time, \( x \) in the above case.
3. Then you write `in`.
4. Then you write an expression. The value that this expression reduces to is what the for loop iterates over. We call it an iterable. In the above case it was \([1, 5, 3, 7]\).
5. The statements to be repeated for every value in the *iterable*.

Try this variation and notice how the rules 1-5 apply in each case:

```python
for x in [1, 5, 3, 7]:
    print(x)

list_of_numbers = [1, 5, 3, 7]
for x in list_of_numbers:
    print(x)

for x in [1, 5] + [3, 7]:
    print(x)
```

In each case the expression after `in` reduces to the value `[1, 5, 3, 7]`, which then serves as the *iterable*.

What is an *iterable*, you may ask? Actually, it is any kind of value that knows how to serve one value at a time until there are none left. Only objects that have an `__iter__` method can do this. If you try to iterate over a value that does not have an `__iter__` method you will get an error. Try the code below and see how Python complains that “‘int’ object is not iterable”:

```python
for x in 4:
    print(x)
```

Not only lists are iterable. Strings are too. Their ‘*iter*’ method of a string tell it that it should serve once character at time. Try this:

```python
for character in 'banana':
    print(character)
```

Neat, right?.

In programming you very often need to iterate over integer values, and sometimes quite a few (like the 250 mill. bases in chromosome one). It would be quite anoying if you had to manually make long lists of integers, so Python provides a builtin function called `range` that helps you out. It returns a special *iterator* value that lets you iterate over a specified range of numbers. Try the two examples below and compare what is printed:

```python
for x in range(1, 5):
    print(x)
```
```python
total = 0
for number in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
    total += number
print(total)

total = 0
for number in range(10):
    total += number
print(total)
```

You can see that using `range` works just like using a list of numbers, but the cool thing about range is that it does not return a list. It just serves one number at a time until it is done. This is also why you will not see a list if you try to print what `range` returns:

```python
number_iterator = range(10)
print(number_iterator)
```

The `range` function needs three values to know which values to iterate over: “start”, “end” and “step”. If you do not give it all three arguments it will assume sensible defaults. Try this:

```python
for i in range(0, 10, 1):
    print(i)

for i in range(0, 10):
    print(i)

for i in range(10):
    print(i)
```

You can see that first and the last arguments default to 0 and 1. If you give it two arguments it will assume that they are “start” and “end”. If you only give it one argument it will assume that it is the “end”.

## Exercises

**Heads up!** In some of the exercises below we mix lists, for-loops and functions. But remember that even though the code may mix lists, for-loops and functions, the *rules* for lists, for-loops and functions are *not* mixed. The separate simple rules for a list, a for-loop and a function are *still* the same. If you get confused, revisit the section on each separate topic.

**Exercise 4.1:** What do you think the third argument to `range` specifies? Try these variations and see if you can figure it out:
for i in range(0, 10, 1):
    print(i)

for i in range(0, 10, 2):
    print(i)

for i in range(0, 10, 3):
    print(i)

Check the documentation once you have made up your mind.

Exercise 4.2: What will happen here:

for x in []:
    print(x)

and here:

for x in range(0):
    print(x)

Exercise 4.3: The two examples below print the same. Make sure you understand why. Write and experiment with the code on your own.

list_of_words = ['one', 'two', 'three']

# example 1
for word in list_of_words:
    print(word)

# example 2
for index in range(len(list_of_words)):
    print(list_of_words[index])

Exercise 4.4: Finish the code below so all the even numbers go into one list and all the odd numbers go into the other (hint: remember the modulo operator?)
numbers = [4, 9, 6, 7, 4, 5, 3, 2, 6]
even = []
odd = []
for n in numbers:
    # your code here ...

Exercise 4.5: Write a function, `square_numbers`, that takes a list of numbers as argument and returns a new list with the numbers squared.

# write your function definition here ...
numbers = [1, 5, 3, 7]

# then you can call it like this:
squared = squared_numbers(numbers)

Exercise 4.6: The code below makes a list of all the `a` characters in banana.

result = []
fruit = 'banana'
for character in fruit:
    if character == 'a':
        result.append(character)
print(result)

Now change the code so you instead get the indexes of the ‘a’ characters: [1, 3, 5]. Here are some hints:

1. You need a for-loop over a list of numbers.
2. `range(len(fruit))` may be relevant numbers :-).

Exercise 4.7: Imagine you want to throw a big party and that you have a rented a place with room for 100 people. Now you want to start inviting people. What kind of error do you get here and why?

friends = ["Mogens", "Preben", "Berit"]
invited = []
for index in range(100):
    invited.append(friends[index])
Creating data structures

In the last exercises of chapter 3 you tried to make some complicated data structures. In this section we will revisit this topic and train your ability to create such data structures.

**Exercise 4.8:** Imagine you want to count how many times each number appears in a DNA string. If you have a DNA string like this: `'ATGCCGATTAA'`, you want to end up with a dictionary like this one (possibly with key-value pairs in a different order):

```
{'G': 2, 'C': 2, 'T': 3, 'A': 4}
```

Here is some code to get you going:

```python
DNA = 'ATGCCGATTAA'
counts = {'G': 0, 'C': 0, 'T': 0, 'A': 0}
for base in DNA:
    # Your code here...
```

**Exercise 4.9:** A similar task is to count how many times each number appears in a list. However, there are infinitely many numbers, so you cannot simply make a key in your dictionary for each number before we start counting. We can solve this problem by only adding keys for the numbers we actually see in the list. To do this we need to change our approach from before in two ways:

1. We start with an empty dictionary, not one initialized with a lot of zero counts.
2. For every number we see we need to check if that number is a key in our dictionary. If it is not, then we need to add it and pair it with the value `0`. You can test if a key is in a dictionary using the `in` operator:

```python
if number not in counts:
    counts[number] = 0
```

Now complete the code below:

```python
counts = {}
number_list = [3, 5, 3, 5, 6, 2]
for number in number_list:
    # Your code here...
```

When you are done you should have a dictionary like this (possibly with key-value pairs in a different order):
Exercise 4.10: The counting technique in the previous exercise lets you count pretty much anything that can be a key in a dictionary. Try using the same approach to count the number of each thing in this list:

stuff = ['sofa', 42, 42, 3.14159, 'sofa', 'Dragon']

Exercise 4.11: Now try to modify the code you made in Exercise 4.9 so that it adds the numbers onto a list instead of incrementing a count. You should end up with a data structure like this:

{2: [2], 3: [3, 3], 5: [5, 5], 6: [6]}

Hint: you only need to make very few changes to the code you already have:

1. Initialize each new key-value pair with [ ] instead of 0.
2. Append each value to that list instead of adding 1 to a number.

Exercise 4.12: Decide what you think the code below does and why you think so. Do every step in your head including all the substitutions and reductions. Then write the code carefully and run it.

nr_list = [10, 20, 30]
combinations = []
for a in nr_list:
    for b in nr_list:
        pair = [a, b]
        combinations.append(pair)

The combinations list becomes:

[[10, 10], [10, 20], [10, 30],
 [20, 10], [20, 20], [20, 30],
 [30, 10], [30, 20], [30, 30]]

Here I broke over three lines to make it fit on the page. You should print your own combinations list to make sure you got the code right.

Exercise 4.13: The code in the exercise above printed all combinations of numbers in the list - including those where the two numbers are the same. Change the code above so these pairs are not printed. You should end up with an list like this:
Exercise 4.13: Can you repeat the previous exercise, by modifying the code below? Begin by making sure you understand why it produces the same list as that in Exercise 4.12. If you have trouble with that, then have another look at Exercise 4.3.

```python	nr_list = [10, 20, 30]
combinations = []
for i in range(len(nr_list)):
    for j in range(len(nr_list)):
        pair = [nr_list[i], nr_list[j]]
        combinations.append(pair)
```

Exercise 4.14: The code in the exercise above printed all combinations of different numbers in the list. But you can see that each pair of numbers still appear twice if you do not take their order into account (e.g. [10, 30] are the same two numbers as [30, 10]). Change the code you wrote for the previous exercise so these pairs are not printed. You should end up with a list like this:

```
[[10, 20], [10, 30], [20, 30]]
```

Hint: the easiest way to do it is to change the range of numbers you iterate over in the second for-loop.

Exercise 4.15: Use the code you made in Exercise 4.9 and change it to make a function `count_characters`, which takes a string argument and returns a dictionary with the counts of each character in the string. So when you call the function like this:

```python
count_characters('banana')
```

it returns:

```
{'n': 2, 'b': 1, 'a': 3}
```

Here is some code to help you along...

```python
def count_characters(text):
    counts = {}
    for character in text:
        # fill in the missing code ...

    return counts
Exercise 4.16: Use the function you made in the previous exercise to construct the following data structure:

```
{  'banana': {'b': 1, 'a': 3, 'n': 2},
    'apple': {'a': 1, 'e': 1, 'p': 2, 'l': 1},
    'ananas': {'a': 3, 's': 1, 'n': 2} }
```

from this list:
```
['banana', 'ananas', 'apple']
```

Here is some code to help you along:
```
my_database = {}
for w in ['banana', 'ananas', 'apple']:
    my_database[w] = # you figure this out...
```

BTW: what do you think  `my_database['banana']['a']` will be subsituted for in an expression?

Exercise 4.17: Sometimes programmers (like you) work with matrices of numbers like the one below:

```
[[1, 2, 3, 4, 5],
 [1, 2, 3, 4, 5],
 [1, 2, 3, 4, 5],
 [1, 2, 3, 4, 5],
 [1, 2, 3, 4, 5]]
```

Here i wrote the list i a nice way so you can see that a matrix is really just a list of lists. When you print it it looks like this:
```
[[1, 2, 3, 4, 5], [1, 2, 3, 4, 5], [1, 2, 3, 4, 5], [1, 2, 3, 4, 5], [1, 2, 3, 4, 5]]
```

Write some code that produces that matrix. If you let out a sigh just now, then quickly reread the sections on lists and for-loops. You may think absorbed all the information you could when you read it the first time, but with more practise you may not be able to understand at deeper level the second or third time you read it.

Exercise 4.18: Can you write some code that produces this matrix?:

```
[[1, 2, 3, 4, 5], [1, 2, 3, 4, 5], [1, 2, 3, 4, 5], [1, 2, 3, 4, 5], [1, 2, 3, 4, 5]]
```
Exercise 4.19: Now produce this matrix:

```
[[1, 1, 1, 1, 1],
 [2, 2, 2, 2, 2],
 [3, 3, 3, 4, 3],
 [4, 4, 5, 4, 4],
 [5, 5, 6, 5]]
```

Exercise 4.20: Can you write some code that produces this matrix?:

```
[[0, 0, 0, 0, 0],
 [0, -1, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0]]
```

Chapter exercises

The exercises in this chapter are more complicated than what you have been exposed to so far and you have probably used most of your brain power at this point. So here is just a single chapter exercise in the end.

Exercise 4.21: Read the code below and make sure you understand each single step before you write any of it. Revisit previous sections if you must, or look in the Python documentation. Then write and run the code - and enjoy that it was exactly what you expected.
def get_words(text, search_string):
    hits = []
    for word in text.split():
        if search_string in word:
            hits.append(word)
    return hits

s = 'eenie meenie minie moe'
nie_words = get_words(s, 'nie')
m_words = get_words(s, 'm')

print(' '.join(nie_words))
print(' '.join(m_words))