# Unit 3: Control flow and list comprehensions

Richard Foltyn

November 26, 2020

## Contents



# 1 Control flow and list comprehensions

In this lecture, we continue to explore basic concepts of the Python programming language such as conditional execution and loops.

## 1.1 Conditional execution

Frequently, we want to execute a code block only if some condition holds. We can do this using the `if` statement:

[1]:
```
if 2*2 == 4:
    print('Python knows arithmetic!')
```

```
Python knows arithmetic!
```

A few observations:

- Conditional blocks are grouped using indentation (leading spaces). Remember from the previous unit that whitespace matters in Python!
- We write the equality operator using *two* equal signs, ==. This is to distinguish it from the assignment operator =.

We can also add an `else` block that will be executed whenever a condition is false:

[2]:
```
if 2*2 == 3:
    # this branch will never be executed
    print('Something is fishy here')
else:
    print('Python knows arithmetic!')
```

```
Python knows arithmetic!
```

Finally, we can add more than one conditional branch using the `elif` clause:

[3]:
```
var = 1
if var == 0:
    print('var is 0')
elif var == 1:
    print('var is 1')
else:
    print('var is neither 0 nor 1')
```

```
var is 1
```

### 1.1.1 Conditional expressions

We already encountered == to test whether two values are equal. Python offers many more operators that return either `True` or `False` and can be used as conditional expressions.

| Expression | Description |
|---|---|
| `==` | Equal. Works for numerical values, strings, etc. |
| `!=` | Not equal. Works on numerical values, strings, etc. |
| `>, >=, <,<=` | Usual comparison of numerical values |
| `a is b, a is not b` | Test identity. `a is b` is `True` if `a` and `b` are the same object (in memory) |
| `a in b, a not in b` | Test whether `a` is or is not included in `b` where `b` is a collection |
| `if obj, if not obj` | Any Python object evaluates to `True` or `False` in an intuitive fashion (see below) |

Additionally, there are logical operators that allow us to combine two logical values:

| Expression | Description |
|---|---|
| `a and b` | `True` if both `a` and `b` are `True` |
| `a or b` | `True` if at least one of `a` or `b` is `True` |

*Examples:*

[4]:
```
# list1 and list2 reference the same object
list1 = [1,2]
list2 = list1
list1 is list2      # objects are identical, returns True
```

[4]: `True`

[5]:
```
# list1 and list2 do NOT reference the same object, but contain
# identical elements.
list2 = list1.copy()
list1 is list2      # returns False
```

[5]: `False`

[6]:
```
# Check if collections contain the same elements
list1 == list2      # returns True
```

[6]: `True`

[7]:
```
# Check whether element is in collection
1 in list1          # returns True
```

[7]:
```
True
```

[8]:
```
# Combine logical expressions using 'and'
1 in list1 and 2 in list1   # returns True
```

[8]:
```
True
```

As mentioned above, any object evaluates to `True` or `False` in an `if` statement:

```
if obj:
    # do something if obj evaluates to True
```

The rules are quite intuitive: an object evaluates to `False` if

- it has a numerical type and is `0` (or `0.0`, or complex `0+0j`)
- it is an empty collection (tuple, list, dictionary, array, etc.)
- it is of logical (boolean) type and has value `False`
- it is `None`, a special built-in value used to denote that a variable does not reference anything.

In all other cases, an expression evaluates to `True`.

## 1.2 Loops

Whenever we want to iterate over several items, we use the `for` loop. The `for` loop in Python is particular powerful because it can "magically" iterate over all sorts of data, not just integer ranges.

The standard use-case is to iterate over a set of integers:

[9]:
```
# iterate over 0,...,3 and print each element
for i in range(4):
    print(i)
```

```
0
1
2
3
```

We use the built-in `range` function to define the sequence of integers over which to loop. As usual in Python, the last element is **not** included. We can explicitly specify the start value and increment using the more advanced syntax `range(start,stop,step)`:

[10]:
```
# iterate over 1,3
for i in range(1,4,2):
    print(i)
```

```
1
3
```

Unlike in some other languages, we can directly iterate over elements of a collection:

[11]:
```
cities = ('Glasgow', 'Edinburgh', 'St. Andrews')
for city in cities:
    print(city)
```

```
Glasgow
Edinburgh
St. Andrews
```

We could of course alternatively iterate over indices and extract the corresponding element, but there is no need to:

[12]:
```
for i in range(len(cities)):
    # print city at index i
    print(cities[i])
```

```
Glasgow
Edinburgh
St. Andrews
```

Sometimes the set of items over which to iterate is not known ex ante, and then we can instead use the `while` loop with a terminal condition:

[13]:
```
z = 1.001
i = 0

# How many iterations will be performed? Not obvious ex ante.
while z < 100.0:
    z = z*z + 0.234
    i = i + 1

# print number of iterations
print("loop terminated after {:d} iterations".format(i))
```

```
loop terminated after 5 iterations
```

### 1.2.1 Advanced looping

Oftentimes, we want to iterate over a list of items and at the same time keep track of an item's index. We can do this elegantly using the `enumerate()` function:

[14]:
```
cities = ('Glasgow', 'Edinburgh', 'St. Andrews')

# Iterate over cities, keep track of index in variable i
for i, city in enumerate(cities):
    print('City {}: {}'.format(i, city))
```

```
City 0: Glasgow
City 1: Edinburgh
City 2: St. Andrews
```

We can skip an iteration or terminate the loop using the `continue` and `break` statements, respectively:

[15]:
```
for city in cities:
    if city == 'Edinburgh':
        # skip to next iteration in case of Edinburgh
        continue
    print(city)
```

```
Glasgow
St. Andrews
```

[16]:
```
for city in cities:
    if city == 'Glasgow':
        # Terminate iteration as soon as we find Glasgow
        print('Found Glasgow')
        break
```

```
Found Glasgow
```

## 1.3 List comprehensions

Python implements a powerful feature called "list comprehensions" that can be used to create collections such as tuples and lists without writing loop statements.

For example, imagine we want to create a list of squares of the integers $0, \ldots, 4$. We can do this using a loop and a list's `append()` method:

[17]:
```
# Initialise empty list
squares = []

# Loop over integers 0,...,4
for i in range(5):
    # The power operator in Python is **
    squares.append(i**2)
squares
```

[17]:
```
[0, 1, 4, 9, 16]
```

This is quite bloated and can be collapsed into a single expression using a list comprehension:

[18]:
```
squares = [i**2 for i in range(5)]
squares
```

[18]:
```
[0, 1, 4, 9, 16]
```

If the desired result should be a `tuple`, we can instead write

[19]:
```
squares = tuple(i**2 for i in range(5))
squares
```

[19]:
```
(0, 1, 4, 9, 16)
```

Alternatively, we can also create a dictionary using curly braces and the syntax `{key: <expression> for ...}`:

[20]:
```
squares = {i: i**2 for i in range(5)}
squares
```

[20]:
```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

List comprehensions can be nested and combined with conditions to create almost arbitrarily complex expressions (this doesn't mean that you should, though!)

[21]:
```
# Create incomprehensible list comprehension.
# The modulo operator in Python is %
items = [i*j for i in range(5) if i % 2 == 0 for j in range(i)]
items
```

[21]:
```
[0, 2, 0, 4, 8, 12]
```

Written out as two nested loops, this code is equivalent to

[22]:
```
items = []
for i in range(5):
    if i % 2 == 0:
        for j in range(i):
            items.append(i*j)
items
```

[22]:
```
[0, 2, 0, 4, 8, 12]
```

# 2 Exercises

These exercises are not meant to demonstrate the most efficient use of Python, but to help you practice the material we have studied above. In fact, you'd most likely *not* want to use the solutions presented here in real code!

## 2.1 Exercise 1: Approximate Euler's number

Euler's number is defined as

$$e = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n$$

1. Create a sequence of the approximations to $e$ for $n = 10, 20, 30, \ldots, 100$ using a list comprehension.
2. Compute the approximation error for each of the elements. *Hint:* To get the built-in value for $e$, use the import statement `from math import e`

## 2.2 Exercise 2: Approximate the sum of a geometric series

Let $\alpha \in (0, 1)$. The infinite sum of the geometric series $(1, \alpha, \alpha^2, \ldots)$ is given by

$$\sigma = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$

Assume that $\alpha = 0.1$. Write a loop that accumulates the values of the sequence $(1, \alpha, \alpha^2, \ldots)$ until the difference to the true value is smaller than $1 \times 10^{-8}$. How many elements does it take?

*Hint:* In Python (and many other languages) the floating-point value $1 \times 10^{-8}$ is written as `1e-8`.

## 2.3 Exercise 3: Binary search [advanced]

The bisection method can be used to find to root of a function $f(x)$, ie. the point $x_0$ such that $f(x_0) = 0$. In this exercise, we will use the same algorithm to find the interval of a strictly monotonic sequence of numbers that brackets the value zero (this is a binary search algorithm with approximate matching).

Assume that we have an array `x` of 11 increasing real numbers given by

[23]:
```
import numpy as np
x = np.linspace(-0.5, 1.0, 11)
x
```

[23]:
```
array([-0.5 , -0.35, -0.2 , -0.05,  0.1 ,  0.25,  0.4 ,  0.55,  0.7 ,
        0.85,  1.  ])
```

Write code that identifies the bracketing interval (in this case [-0.05,0.1]) using the following algorithm:

1. Initialize the index of the bracket lower bound to `lbound=0` and the index of the bracket upper bound to `ubound=len(x)-1`.
2. Compute the midpoint between these two indices (rounded to the nearest integer), `mid = (ubound + lbound) // 2`. *Hint:* The operator `//` truncates the result of a division to the nearest integer.
3. Inspect `x[mid]`, the value at index `mid`. If `x[mid]` has the same sign as `x[ubound]`, update the upper bound, `ubound=mid`. Otherwise, update the lower bound.
4. Continue until `ubound = lbound + 1`, ie. until you have found the bracket `x[lbound] <= 0 < x[ubound]`.

# 3 Solutions

## 3.1 Solution for exercise 1

[24]:
```
# Compute approximation for n = 10, 20,..., 100
euler_approx = [(1.0+1.0/i)**i for i in range(10,101,10)]
print('Approximate values')
print(euler_approx)

# import 'correct' value
from math import e

# We need to subtract e from each element to get the approximation error
euler_error = [approx - e for approx in euler_approx]
print('Approximation error')
print(euler_error)
```

```
Approximate values
[2.5937424601000023, 2.653297705144422, 2.6743187758703026, 2.685063838389963,
2.691588029073608, 2.6959701393302162, 2.6991163709761854, 2.7014849407533275,
2.703332461058186, 2.7048138294215285]
Approximation error
[-0.12453936835904278, -0.06498412331462289, -0.043963052588742446,
-0.03321799006908188, -0.026693799385437256, -0.02231168912882886,
-0.019165457482859694, -0.016796887705717634, -0.01494936740085917,
-0.01346799903751661]
```

## 3.2 Solution for exercise 2

We don't know now many iterations we will need to get to the required tolerance of $1 \times 10^{-8}$, so this is a good opportunity to use a `while` loop.

[25]:
```
# Convergence tolerance
tol = 1e-8
alpha = 0.1
# The correct value
sigma_exact = 1.0/(1.0 - alpha)

# keep track of number of iterations
n = 0

# Initialise approximated sum
sigma = 0.0

# Iterate until absolute difference is smaller than tolerance level.
# The built-in function abs() returns the absolute value.

while abs(sigma - sigma_exact) > tol:
    # We can combine addition and assignment into a single operator +=
    # This is equivalent to
    #   sigma = sigma + alpha**n
    sigma += alpha**n
    # Increment exponent
    n += 1

print('Number of iterations: {}, approx. sum: {:.8f}'.format(n, sigma))
```

```
Number of iterations: 9, approx. sum: 1.11111111
```

### 3.3 Solution for exercise 3

To complete the exercise, all you have to do is to translate the algorithm given in the exercise into code.

Since we don't know how many iterations it takes to find the bracket, we use a `while` loop that continues as along as `lbound` and `ubound` are more than 1 apart.

[26]:
```
import numpy as np

# Given array of increasing numbers
x = np.linspace(-0.5, 1.0, 11)


lbound = 0
ubound = len(x) - 1

while ubound > (lbound + 1):
    # Index of mid point. Indices have to be integers, so
    # we need to truncate the division result to an integer.
    mid = (ubound + lbound) // 2

    if (x[mid] * x[ubound]) > 0.0:
        # x[mid] and x[ubound] have same sign!
        ubound = mid
        print('Setting upper bound index to {}'.format(ubound))
    else:
        # x[mid] and x[lbound] have the same sign
        # or at least one of them is zero
        lbound = mid
        print('Setting lower bound index to {}'.format(lbound))

print('Value at lower bound: {:.4g}'.format(x[lbound]))
print('Value at upper bound: {:.4g}'.format(x[ubound]))
```

```
Setting upper bound index to 5
Setting lower bound index to 2
Setting lower bound index to 3
Setting upper bound index to 4
Value at lower bound: -0.05
Value at upper bound: 0.1
```

In this implementation we use the fact that two non-zero real numbers $a$ and $b$ have the same sign whenever $a \cdot b > 0$.