Lecture 2: Control flow and list comprehensions

FIE463: Numerical Methods in Macroeconomics and Finance using Python

Richard Foltyn
NHH Norwegian School of Economics

January 27, 2026

See GitHub repository for notebooks and data:

https://github.com/richardfoltyn/FIE463-V26

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 Truth value testing

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 to control conditional execution.

Expression	Description
`==`	Equal. Works for numerical values, strings, etc.
`!=`	Not equal. Works for 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
`a in b, a not in b`	Test whether `a` is or is not included in collection `b`
`if obj, if not obj`	Most Python objects evaluate to `True` or `False` in an intuitive fashion (see below)

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

Expression	Description
`not a`	Invert the logical value of `a` (`True` if `a` is `False`)
`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]:
```
# a and b reference the same object
a = [1, 2]
b = a
a is b          # objects are identical, returns True
```

[4]:
```
True
```

[5]:
```
# a and b do NOT reference the same object, but contain
# identical elements.
b = a.copy()        # Use copy() method to create a copy
a is b              # returns False
```

[5]:
```
False
```

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

[6]: True

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

[7]: True

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

[8]: True

We can also use the `in` operator to determine whether a key is contained in a dictionary:

[9]:
```
dct = {'institution': 'NHH'}
'institution' in dct        # prints True
```

[9]: True

[10]:
```
# check whether a key is NOT in the dictionary
'course' not in dct         # prints True
```

[10]: True

As mentioned above, most objects evaluate 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 string `''`
- it is an empty collection (tuple, list, dictionary, 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 most other cases, an expression evaluates to `True`.

Examples:

[11]:
```
# evaluate numerical variable
x = 0.0
if x:
    print('Value is non-zero')
else:
    print('Value is zero')
```

```
Value is zero
```

[12]:
```
# Evaluate non-empty string
x = 'Hello'
if x:
    print('String is non-empty')
else:
    print('String is empty')
```

```
String is non-empty
```

[13]:
```
# Check whether a collection (tuple, list, dictionary) is empty
# Create empty dictionary
dct = {}
if dct:
    print('Dictionary is non-empty')
else:
    print('Dictionary is empty')
```

```
Dictionary is empty
```

[14]:
```
# Evaluate boolean variable
flag = True
if flag:
    print('Flag is True')
else:
    print('Flag is False')
```

```
Flag is True
```

The most important exception to the rule that objects intuitively evaluate to `True` or `False` is the behavior of NumPy arrays:

[15]:
```
import numpy as np

# Create array with 5 zeros
arr = np.zeros(5)
if arr:
    print('true!')
```

```
ValueError: The truth value of an array with more than one element is ambiguous. Use a.any )
 ↪or a.all()
```

As the error message indicates, NumPy requires you to be more specific about what exactly should be tested.

Your turn.

Check whether it is possible to perform the following comparisons and inspect their results.

1. Define a list `[1.0, 2.0]` and a tuple `(1.0, 2.0)`, and check for equality using `==`.
2. Define a list `[1.0, 2.0]` and a NumPy array with elements `[1.0, 2.0]`, and check for equality using `==`.
3. What does the expression `'P' in 'Python'` evaluate to? How about `'Py' in 'Python'`?
4. What does the expression `'a' in ['a', 'b']` evaluate to? How about `['a'] in ['a', 'b']`?
5. What is the value of the expression `1.0 == (1.0 + 1.0e-16)`?

1.1.2 Conditional expressions

Conditional expressions are more compact than conditional statements and can be used to return a value depending on some condition. A conditional expression takes three arguments using the syntax

```
<value if true> if <condition> else <value if false>
```

The value of this expression can be assigned to a variable, passed to a function, etc.

To illustrate, imagine we have the following code:

[16]:
```
x = 1

# Test whether x is divisible by 2 without remainder using
# the modulo operator %
if (x % 2) == 0:
    var = 'even'
else:
    var = 'odd'
```

This code sets the value of `var` to either `'even'` or `'odd'`, depending on whether `x` is an even or odd integer. We can formulate this more concisely using a conditional expression:

[17]:
```
var = 'even' if (x % 2) == 0 else 'odd'
```

We can even directly print the value of this expression!

[18]:
```
x = 1
print('even' if (x % 2) == 0 else 'odd')
```

```
odd
```

There is an even more compact notation that is often used to set a default value. Say that you write a function which takes a string argument and stores its value in the variable `argument` (we will study functions in the next lecture). This argument can potentially be an empty string, and in that case, you want to use a default value. In the function body, you could write code such as the following to achieve this:

[19]:
```
# Assume empty string is passed as argument by the caller of a function
argument = ''
# Default value
default = 'default value'

# Check whether argument is empty, and if so, use the default value
if argument:
    var = argument
else:
    var = default

var
```

[19]:
```
'default value'
```

This is admissible code but needlessly complicated. Using conditional expressions, you can abbreviate this as follows:

[20]:
```
# Set default value if argument is empty
var = argument if argument else default

var
```

[20]:
```
'default value'
```

This of course works as well, but there is an even shorter version using the logical `or` operator:

[21]:
```
var = argument or default

var
```

[21]:
```
'default value'
```

This logical `or` operator works as follows: if the left-hand-side operand evaluates to `True` (as a non-empty string does), the evaluation is aborted and this left-hand-side value is returned. If the left-hand-side operand evaluates to `False`, the right-hand-side operand is returned.

1.2 Loops

1.2.1 The `for` loop

Whenever we want to iterate over several items, we use the `for` loop. The `for` loop in Python is particularly 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:

[22]:
```
# 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)`:

[23]:
```
# iterate over 1,3
for i in range(1, 4, 2):
    print(i)
```

```
1
3
```

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

[24]:
```
cities = ('Bergen', 'Oslo', 'Stavanger')
for city in cities:
    print(city)
```

```
Bergen
Oslo
Stavanger
```

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

[25]:
```
# Needlessly complicated and non-Pythonic
cities = ('Bergen', 'Oslo', 'Stavanger')
for i in range(len(cities)):
    # print city at index i
    print(cities[i])
```

```
Bergen
Oslo
Stavanger
```

Your turn.

Write a `for` loop that prints the squares of all even numbers between 0 and 10 (inclusive). *Hint:* The exponentiation operator in Python is `**`.

Looping over dictionaries

When looping over dictionaries, the default is to iterate over *keys*:

[26]:
```
dct = {'key1': 'value1', 'key2': 'value2'}
for key in dct:
    print(key)
```

```
key1
key2
```

We can explicitly choose whether to iterate over keys, values, or both:

[27]:
```
# iterate over keys, same as example above
for key in dct.keys():
    print(key)
```

```
key1
key2
```

[28]:
```
# iterate over values
for value in dct.values():
    print(value)
```

```
value1
value2
```

[29]:
```
# iterate over keys and values at the same time
# using the items() method
for key, value in dct.items():
    # use f-string to print key: value
    print(f'{key}: {value}')
```

```
key1: value1
key2: value2
```

Note that `items()` returns the key-value pairs as `tuples`, so we need to unpack each tuple by writing `key, value` as the running variables of the `for` loop.

1.2.2 The `while` loop

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 termination condition:

[30]:
```
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(f'loop terminated after {i} iterations; z = {z}')
```

```
loop terminated after 5 iterations; z = 129.58937197374684
```

Your turn.

Use the `while` loop to emulate a `for` loop: Iterate over the set of integers 0,...,4 and print the loop variable in each iteration.

1.2.3 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:

[31]:
```
cities = ('Bergen', 'Oslo', 'Stavanger')

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

```
City 0: Bergen
City 1: Oslo
City 2: Stavanger
```

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

[32]:
```
for city in cities:
    if city == 'Oslo':
        # skip to next iteration in case of Oslo
        continue
    print(city)
```

```
Bergen
Stavanger
```

[33]:
```
for city in cities:
    if city == 'Bergen':
        # Terminate iteration as soon as we find Bergen
        print(f'Found {city}')
        break
```

```
Found Bergen
```

We often want to loop over multiple collections at once. This can be achieved in various ways, but if we are working with collections of equal length, we usually use the `zip()` function.

Imagine we are given a list of regions and their capital cities and want to print each region together with its capital:

[34]:
```
# List of capitals
capitals = ['Bergen', 'Stavanger', 'Trondheim']
# List of regions
regions = ['Vestland', 'Rogaland', 'Trøndelag']

# Iterate through capitals and regions in parallel
for c, r in zip(capitals, regions):
    print(f'The capital of {r} is {c}')
```

```
The capital of Vestland is Bergen
The capital of Rogaland is Stavanger
The capital of Trøndelag is Trondheim
```

In other programming languages, we would have likely used a running index instead, but this is unnecessary in Python:

[35]:
```
# Alternative, non-Pythonic implementation
for i in range(len(capitals)):
    print(f'The capital of {regions[i]} is {capitals[i]}')
```

```
The capital of Vestland is Bergen
The capital of Rogaland is Stavanger
The capital of Trøndelag is Trondheim
```

Your turn.

Consider the following list of Nordic countries and their capitals:
`capitals = ['Oslo', 'Stockholm', 'Copenhagen', 'Helsinki']`
`countries = ['Norway', 'Sweden', 'Denmark', 'Finland']`
Using a `for` loop, create a dictionary that maps the country (the key) to its capital (the value).

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:

[36]:
```
# Initialize empty list
squares = []

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

[36]:
```
[0, 1, 4, 9, 16]
```

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

[37]:
```
squares = [i**2 for i in range(5)]
squares
```

[37]:
```
[0, 1, 4, 9, 16]
```

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

[38]:
```
squares = tuple(i**2 for i in range(5))
squares
```

[38]:
```
(0, 1, 4, 9, 16)
```

Note that the `tuple()` syntax is mandatory: one cannot create a tuple using a list comprehension with only `()` as this returns a generator object instead:

[39]:
```
# Wrong attempt to create a tuple from a list comprehension, returns generator
squares = (i**2 for i in range(5))
squares
```

[39]:
```
<generator object <genexpr> at 0x7fab10135ff0>
```

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

[40]:
```
squares = {i: i**2 for i in range(5)}
squares
```

[40]:
```
{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!)

[41]:
```
# 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
```

[41]:
```
[0, 2, 0, 4, 8, 12]
```

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

[42]:
```
items = []
for i in range(5):
    if i % 2 == 0:
        for j in range(i):
            items.append(i*j)
items
```

[42]:
```
[0, 2, 0, 4, 8, 12]
```

Your turn.

Adapt the previous exercise mapping countries to their capitals: instead of using a loop, create the dictionary mapping countries to their capitals using a *list comprehension*.

1.4 List of functions used in this lecture

The following list contains the central functions covered in this lecture. Each function name is linked to the official API documentation which you can consult for more details regarding function arguments and return values. The [go to section] links to the relevant section in this notebook.

1.4.1 Python built-in functions

- `range()`: Function for generating a sequence of numbers. [go to section]
- `enumerate()`: Return an enumerate object yielding pairs of index and value. [go to section]
- `zip()`: Iterate over several iterables in parallel. [go to section]

1.4.2 NumPy array creation functions

- `zeros()`: Return a new array of given shape and type, filled with zeros. [go to section]

1.5 Optional 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!

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 and print 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
   ```

Exercise 2: Approximate the sum of a geometric series

Let $\alpha \in (0, 1)$. The sum of the geometric series $(1, \alpha, \alpha^2, \dots)$ 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 $(1, \alpha, \alpha^2, \dots)$ until the difference to the true value is smaller than 1×10^{-8}. How many iterations does it take?

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

Exercise 3: Diagonal and band matrices

Let `a` be a matrix of zeros with shape `(m,n)` with an integer data type:

```
a = np.zeros((m,n), dtype=int)
```

1. Set `m = n = 4`. Fill up the diagonal of `a` with ones so that it becomes an identity matrix. Verify that `np.identity()` gives the same result.
2. Create a 4×5 matrix `b` of zeros. Modify the main, first lower and first upper diagonals so that the resulting matrix looks like this, where omitted elements are zeros:

$$\begin{bmatrix} 1 & 2 & & & \\ 3 & 1 & 2 & & \\ & 3 & 1 & 2 & \\ & & 3 & 1 & 2 \end{bmatrix}$$

   Hint: Use nested `for` loops to set the diagonal elements.

Exercise 4: Triangular matrices

Let `a` be a matrix of zeros with shape `(m,n)` and integer data type:

```
a = np.zeros((m,n), dtype=int)
```

Assume that `m = n = 4`. Using loops and `if` statements, modify the elements of `a` to obtain the following upper-triangular matrix, where omitted elements are set to zero:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ & 5 & 6 & 7 \\ & & 8 & 9 \\ & & & 10 \end{bmatrix}$$

Exercise 5: Binary search (advanced)

The bisection method can be used to find the root of a function $f(x)$, i.e., 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

[43]:
```
import numpy as np
x = np.linspace(-0.5, 1.0, 11)
x
```

[43]:
```
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`, i.e., until you have found the bracket `x[lbound] <= 0 < x[ubound]`.

1.6 Solutions

Solution for exercise 1

[44]:
```
# 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]
```

Solution for exercise 2

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

[45]:
```
# 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

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

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

Solution for exercise 3

For the first part, we need to loop over the diagonal elements and overwrite the zeros with ones:

[46]:
```
# Part 1: create identity matrix

import numpy as np

n = 4
a = np.zeros((n,n), dtype=int)

# loop over diagonal elements, set them to 1
for i in range(n):
    a[i,i] = 1

# print identity matrix
a
```

[46]:
```
array([[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 1, 0],
       [0, 0, 0, 1]])
```

You can get the same result using NumPy's `np.identity()` function:

[47]:
```
n = 4
np.identity(n, dtype=int)
```

[47]:
```
array([[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 1, 0],
       [0, 0, 0, 1]])
```

For the second part, we need nested loops over rows and columns and we check at each position (i, j) whether it is on the main diagonal, or on the first upper or lower diagonal.

[48]:
```
# Part 2: create band matrix

import numpy as np

m = 4
```

```
n = 5

b = np.zeros((m,n), dtype=int)

# loop over rows
for i in range(m):
    # loop over columns
    for j in range(n):
        if i == j:
            # main diagonal element
            b[i,j] = 1
        elif i == (j - 1):
            # upper diagonal element
            b[i,j] = 2
        elif i == (j + 1):
            # lower diagonal element
            b[i,j] = 3

# print final matrix
b
```

[48]:
```
array([[1, 2, 0, 0, 0],
       [3, 1, 2, 0, 0],
       [0, 3, 1, 2, 0],
       [0, 0, 3, 1, 2]])
```

Solution for exercise 4

We solve this exercise using two nested `for` loops, the first over rows, the second over columns:

[49]:
```
import numpy as np

m = 4
n = 4

a = np.zeros((m,n), dtype=int)

# keep track of value to be inserted
value = 1

# loop over rows
for i in range(m):
    # loop over columns
    for j in range(i, n):
        a[i,j] = value
        # increment value for next matching element
        value += 1

# print final matrix
a
```

[49]:
```
array([[ 1,  2,  3,  4],
       [ 0,  5,  6,  7],
       [ 0,  0,  8,  9],
       [ 0,  0,  0, 10]])
```

Note that the loop over columns uses the current row index as the lower bound, since we never need to insert anything at element (i, j) if $j < i$.

Solution for exercise 5

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 long as `lbound` and `ubound` are more than 1 apart.

[50]:
```
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(f'Setting upper bound index to {ubound}')
    else:
        # x[mid] and x[lbound] have the same sign
        # or at least one of them is zero
        lbound = mid
        print(f'Setting lower bound index to {lbound}')

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

```
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 $a \cdot b > 0$.