1 Contents

- Overview 2
- Data Types 3
- Input and Output 4
- Iterating 5
- Comparisons and Logical Operators 6
- More Functions 7
- Coding Style and PEP8 8
- Exercises 9
- Solutions 10

2 Overview

We have covered a lot of material quite quickly, with a focus on examples. Now let’s cover some core features of Python in a more systematic way. This approach is less exciting but helps clear up some details.

3 Data Types

Computer programs typically keep track of a range of data types. For example, 1.5 is a floating point number, while 1 is an integer.

Programs need to distinguish between these two types for various reasons. One is that they are stored in memory differently. Another is that arithmetic operations are different

- For example, floating point arithmetic is implemented on most machines by a specialized Floating Point Unit (FPU).

In general, floats are more informative but arithmetic operations on integers are faster and more accurate.

Python provides numerous other built-in Python data types, some of which we’ve already met
• strings, lists, etc.
Let’s learn a bit more about them.

3.1 Primitive Data Types

One simple data type is **Boolean values**, which can be either **True** or **False**

```
In [1]: x = True
x
```

```
Out[1]: True
```

We can check the type of any object in memory using the `type()` function.

```
In [2]: type(x)
```

```
Out[2]: bool
```

In the next line of code, the interpreter evaluates the expression on the right of = and binds `y` to this value

```
In [3]: y = 100 < 10
y
```

```
Out[3]: False
```

```
In [4]: type(y)
```

```
Out[4]: bool
```

In arithmetic expressions, **True** is converted to `1` and **False** is converted `0`.

This is called **Boolean arithmetic** and is often useful in programming.

Here are some examples

```
In [5]: x + y
```

```
Out[5]: 1
```

```
In [6]: x * y
```

```
Out[6]: 0
```

```
In [7]: True + True
```

```
Out[7]: 2
```
Complex numbers are another primitive data type in Python

```python
In [9]: x = complex(1, 2)
y = complex(2, 1)
print(x * y)
type(x)
```

```
5j
Out[9]: complex
```

### 3.2 Containers

Python has several basic types for storing collections of (possibly heterogeneous) data. We’ve already discussed lists.

A related data type is tuples, which are “immutable” lists

```python
In [10]: x = ('a', 'b')  # Parentheses instead of the square brackets
    x = 'a', 'b'  # Or no brackets --- the meaning is identical
    x
```

```
Out[10]: ('a', 'b')
```

```python
In [11]: type(x)
```

```
Out[11]: tuple
```

In Python, an object is called immutable if, once created, the object cannot be changed. Conversely, an object is mutable if it can still be altered after creation.

Python lists are mutable

```python
In [12]: x = [1, 2]
x[0] = 10
x
```

```
Out[12]: [10, 2]
```

But tuples are not
In [13]: x = (1, 2)
x[0] = 10

---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-13-d1b2647f6c81> in <module>
     1 x = (1, 2)
     2 x[0] = 10

TypeError: 'tuple' object does not support item assignment

We’ll say more about the role of mutable and immutable data a bit later.

Tuples (and lists) can be “unpacked” as follows

In [14]: integers = (10, 20, 30)
x, y, z = integers
x

Out[14]: 10

In [15]: y

Out[15]: 20

You’ve actually seen an example of this already.

Tuple unpacking is convenient and we’ll use it often.

3.2.1 Slice Notation

To access multiple elements of a list or tuple, you can use Python’s slice notation.

For example,

In [16]: a = [2, 4, 6, 8]
a[1:]

Out[16]: [4, 6, 8]

In [17]: a[1:3]

Out[17]: [4, 6]
The general rule is that $a[m:n]$ returns $n - m$ elements, starting at $a[m]$. Negative numbers are also permissible.

In [18]: a[-2:]  # Last two elements of the list
Out[18]: [6, 8]

The same slice notation works on tuples and strings

In [19]: s = 'foobar'
s[-3:]  # Select the last three elements
Out[19]: 'bar'

3.2.2 Sets and Dictionaries

Two other container types we should mention before moving on are sets and dictionaries. Dictionaries are much like lists, except that the items are named instead of numbered.

In [20]: d = {'name': 'Frodo', 'age': 33}
   type(d)
Out[20]: dict

In [21]: d['age']
Out[21]: 33

The names 'name' and 'age' are called the keys. The objects that the keys are mapped to ("Frodo" and 33) are called the values.

Sets are unordered collections without duplicates, and set methods provide the usual set-theoretic operations

In [22]: s1 = {'a', 'b'}
   type(s1)
Out[22]: set

In [23]: s2 = {'b', 'c'}
   s1.issubset(s2)
Out[23]: False

In [24]: s1.intersection(s2)
Out[24]: {'b'}

The set() function creates sets from sequences

In [25]: s3 = set(('foo', 'bar', 'foo'))
   s3
Out[25]: {'bar', 'foo'}
4 Input and Output

Let’s briefly review reading and writing to text files, starting with writing

```python
In [26]: f = open('newfile.txt', 'w')  # Open 'newfile.txt' for writing
   f.write('Testing\n')  # Here '\n' means new line
   f.write('Testing again')
   f.close()
```

Here

- The built-in function `open()` creates a file object for writing to.
- Both `write()` and `close()` are methods of file objects.

Where is this file that we’ve created?

Recall that Python maintains a concept of the present working directory (pwd) that can be located from with Jupyter or IPython via

```bash
In [27]: %pwd
```

```
Out[27]: '/home/ubuntu/repos/lecture-source-py/_build/jupyterpdf/executed'
```

If a path is not specified, then this is where Python writes to.

We can also use Python to read the contents of `newline.txt` as follows

```python
In [28]: f = open('newfile.txt', 'r')
   out = f.read()
   out
```

```
Out[28]: 'Testing\nTesting again'
```

```python
In [29]: print(out)
```

```
Testing
Testing again
```

4.1 Paths

Note that if `newfile.txt` is not in the present working directory then this call to `open()` fails.

In this case, you can shift the file to the pwd or specify the full path to the file

```python
f = open('insert_full_path_to_file/newfile.txt', 'r')
```
5 Iterating

One of the most important tasks in computing is stepping through a sequence of data and performing a given action.

One of Python’s strengths is its simple, flexible interface to this kind of iteration via the for loop.

5.1 Looping over Different Objects

Many Python objects are “iterable”, in the sense that they can be looped over.

To give an example, let’s write the file us_cities.txt, which lists US cities and their population, to the present working directory.

In [30]: %file us_cities.txt
    
    new york: 8244910
    los angeles: 3819702
    chicago: 2707120
    houston: 2145146
    philadelphia: 1536471
    phoenix: 1469471
    san antonio: 1359758
    san diego: 1326179
    dallas: 1223229

    Overwriting us_cities.txt

Here %%file is an IPython cell magic.

Suppose that we want to make the information more readable, by capitalizing names and adding commas to mark thousands.

The program below reads the data in and makes the conversion:

In [31]: data_file = open('us_cities.txt', 'r')
    
    for line in data_file:
        city, population = line.split(':')   # Tuple unpacking
        city = city.title()                 # Capitalize city names
        population = f'{int(population):,}' # Add commas to numbers
        print(city.ljust(15) + population)
    data_file.close()

    New York     8,244,910
    Los Angeles  3,819,702
    Chicago      2,707,120
    Houston      2,145,146
    Philadelphia 1,536,471
    Phoenix      1,469,471
    San Antonio  1,359,758
    San Diego    1,326,179
    Dallas       1,223,229
Here `format()` is a string method used for inserting variables into strings.

The reformatting of each line is the result of three different string methods, the details of which can be left till later.

The interesting part of this program for us is line 2, which shows that

1. The file object `data_file` is iterable, in the sense that it can be placed to the right of `in` within a `for` loop.
2. Iteration steps through each line in the file.

This leads to the clean, convenient syntax shown in our program.

Many other kinds of objects are iterable, and we’ll discuss some of them later on.

### 5.2 Looping without Indices

One thing you might have noticed is that Python tends to favor looping without explicit indexing.

For example,

```python
In [32]: x_values = [1, 2, 3]  # Some iterable x
   ...: for x in x_values:
   ...:     print(x * x)
```

```
1
4
9
```

is preferred to

```python
In [33]: for i in range(len(x_values)):
   ...:     print(x_values[i] * x_values[i])
```

```
1
4
9
```

When you compare these two alternatives, you can see why the first one is preferred.

Python provides some facilities to simplify looping without indices.

One is `zip()`, which is used for stepping through pairs from two sequences.

For example, try running the following code

```python
In [34]: countries = ['Japan', 'Korea', 'China']
         cities = ['Tokyo', 'Seoul', 'Beijing']
         for country, city in zip(countries, cities):
             print(f'The capital of {country} is {city}"
```
The capital of Japan is Tokyo
The capital of Korea is Seoul
The capital of China is Beijing

The \texttt{zip()} function is also useful for creating dictionaries — for example:

\begin{verbatim}
In [35]: names = ['Tom', 'John']
   marks = ['E', 'F']
   dict(zip(names, marks))
Out[35]: {'Tom': 'E', 'John': 'F'}
\end{verbatim}

If we actually need the index from a list, one option is to use \texttt{enumerate()}. To understand what \texttt{enumerate()} does, consider the following example:

\begin{verbatim}
In [36]: letter_list = ['a', 'b', 'c']
   for index, letter in enumerate(letter_list):
       print(f"letter_list[{index}] = '{letter}'")

   letter_list[0] = 'a'
   letter_list[1] = 'b'
   letter_list[2] = 'c'
\end{verbatim}

5.3 List Comprehensions

We can also simplify the code for generating the list of random draws considerably by using something called a \textit{list comprehension}.

\textbf{List comprehensions} are an elegant Python tool for creating lists.

Consider the following example, where the list comprehension is on the right-hand side of the second line:

\begin{verbatim}
In [37]: animals = ['dog', 'cat', 'bird']
   plurals = [animal + 's' for animal in animals]
   plurals
Out[37]: ['dogs', 'cats', 'birds']
\end{verbatim}

Here's another example:

\begin{verbatim}
In [38]: range(8)
Out[38]: range(0, 8)
In [39]: doubles = [2 * x for x in range(8)]
   doubles
Out[39]: [0, 2, 4, 6, 8, 10, 12, 14]
\end{verbatim}
6 Comparisons and Logical Operators

6.1 Comparisons

Many different kinds of expressions evaluate to one of the Boolean values (i.e., True or False).

A common type is comparisons, such as

\[
\begin{align*}
\text{In [40]:} & \quad x, y = 1, 2 \\
& \quad x < y \\
\text{Out[40]:} & \quad \text{True} \\
\text{In [41]:} & \quad x > y \\
\text{Out[41]:} & \quad \text{False}
\end{align*}
\]

One of the nice features of Python is that we can chain inequalities

\[
\begin{align*}
\text{In [42]:} & \quad 1 < 2 < 3 \\
\text{Out[42]:} & \quad \text{True} \\
\text{In [43]:} & \quad 1 <= 2 <= 3 \\
\text{Out[43]:} & \quad \text{True}
\end{align*}
\]

As we saw earlier, when testing for equality we use ==

\[
\begin{align*}
\text{In [44]:} & \quad x = 1 \quad # \text{Assignment} \\
& \quad x == 2 \quad # \text{Comparison} \\
\text{Out[44]:} & \quad \text{False}
\end{align*}
\]

For “not equal” use !=

\[
\begin{align*}
\text{In [45]:} & \quad 1 != 2 \\
\text{Out[45]:} & \quad \text{True}
\end{align*}
\]

Note that when testing conditions, we can use any valid Python expression

\[
\begin{align*}
\text{In [46]:} & \quad x = 'yes' \text{ if } 42 \text{ else } 'no' \\
& \quad x \\
\text{Out[46]:} & \quad 'yes'
\end{align*}
\]
What’s going on here?

The rule is:

- Expressions that evaluate to zero, empty sequences or containers (strings, lists, etc.) and None are all equivalent to False.
  - for example, [] and () are equivalent to False in an if clause
- All other values are equivalent to True.
  - for example, 42 is equivalent to True in an if clause

6.2 Combining Expressions

We can combine expressions using and, or and not.
These are the standard logical connectives (conjunction, disjunction and denial)

Remember

- P and Q is True if both are True, else False
- P or Q is False if both are False, else True

7 More Functions

Let’s talk a bit more about functions, which are all important for good programming style.
7.1 The Flexibility of Python Functions

As we discussed in the previous lecture, Python functions are very flexible.

In particular

- Any number of functions can be defined in a given file.
- Functions can be (and often are) defined inside other functions.
- Any object can be passed to a function as an argument, including other functions.
- A function can return any kind of object, including functions.

We already gave an example of how straightforward it is to pass a function to a function.

Note that a function can have arbitrarily many return statements (including zero).

Execution of the function terminates when the first return is hit, allowing code like the following example

```python
In [53]: def f(x):
    if x < 0:
        return 'negative'
    return 'nonnegative'
```

Functions without a return statement automatically return the special Python object None.

7.2 Docstrings

Python has a system for adding comments to functions, modules, etc. called docstrings.

The nice thing about docstrings is that they are available at run-time.

Try running this

```python
In [54]: def f(x):
   """
   This function squares its argument
   """
   return x**2
```

After running this code, the docstring is available

```python
In [55]: f?
```

Type: function
String Form:<function f at 0x2223320>
File: /home/john/temp/temp.py
Definition: f(x)
Docstring: This function squares its argument

```python
In [56]: f??
```
7.3 One-Line Functions: \texttt{lambda}

The \texttt{lambda} keyword is used to create simple functions on one line.

For example, the definitions

\begin{verbatim}
In [57]: def f(x):
    \texttt{return} x**3
\end{verbatim}

and

\begin{verbatim}
In [58]: f = \texttt{lambda} x: x**3
\end{verbatim}

are entirely equivalent.

To see why \texttt{lambda} is useful, suppose that we want to calculate $\int_0^2 x^3 dx$ (and have forgotten our high-school calculus).

The SciPy library has a function called \texttt{quad} that will do this calculation for us.

The syntax of the \texttt{quad} function is \texttt{quad(f, a, b)} where \texttt{f} is a function and \texttt{a} and \texttt{b} are numbers.

To create the function $f(x) = x^3$ we can use \texttt{lambda} as follows

\begin{verbatim}
In [59]: \texttt{from scipy.integrate import quad}
quad(\texttt{lambda} x: x**3, 0, 2)
\end{verbatim}

\texttt{Out[59]}: (4.0, 4.440892098500626e-14)

Here the function created by \texttt{lambda} is said to be \textit{anonymous} because it was never given a name.
7.4 Keyword Arguments

In a previous lecture, you came across the statement

```python
plt.plot(x, 'b-', label="white noise")
```

In this call to Matplotlib’s `plot` function, notice that the last argument is passed in `name=argument` syntax.

This is called a **keyword argument**, with `label` being the keyword.

Non-keyword arguments are called **positional arguments**, since their meaning is determined by order

- `plot(x, 'b-', label="white noise")` is different from `plot('b-', x, label="white noise")`

Keyword arguments are particularly useful when a function has a lot of arguments, in which case it’s hard to remember the right order.

You can adopt keyword arguments in user-defined functions with no difficulty.

The next example illustrates the syntax

```python
In [60]: def f(x, a=1, b=1):
    \n    return a + b * x
```

The keyword argument values we supplied in the definition of `f` become the default values

```python
In [61]: f(2)
```

**Out[61]:** 3

They can be modified as follows

```python
In [62]: f(2, a=4, b=5)
```

**Out[62]:** 14

8 Coding Style and PEP8

To learn more about the Python programming philosophy type `import this` at the prompt.

Among other things, Python strongly favors consistency in programming style.

We’ve all heard the saying about consistency and little minds.

In programming, as in mathematics, the opposite is true

- A mathematical paper where the symbols $\cup$ and $\cap$ were reversed would be very hard to read, even if the author told you so on the first page.

In Python, the standard style is set out in **PEP8**.

(Occasionally we’ll deviate from PEP8 in these lectures to better match mathematical notation)
9 Exercises

Solve the following exercises.

(For some, the built-in function \texttt{sum()} comes in handy).

9.1 Exercise 1

Part 1: Given two numeric lists or tuples \texttt{x_vals} and \texttt{y_vals} of equal length, compute their inner product using \texttt{zip()}.

Part 2: In one line, count the number of even numbers in 0,…,99.

\begin{itemize}
  \item Hint: \texttt{x \% 2} returns 0 if \texttt{x} is even, 1 otherwise.
\end{itemize}

Part 3: Given \texttt{pairs = ((2, 5), (4, 2), (9, 8), (12, 10))}, count the number of pairs \texttt{(a, b)} such that both \texttt{a} and \texttt{b} are even.

9.2 Exercise 2

Consider the polynomial

\[ p(x) = a_0 + a_1 x + a_2 x^2 + \cdots a_n x^n = \sum_{i=0}^{n} a_i x^i \]  

Write a function \texttt{p} such that \texttt{p(x, coeff)} that computes the value in \texttt{(1)} given a point \texttt{x} and a list of coefficients \texttt{coeff}.

Try to use \texttt{enumerate()} in your loop.

9.3 Exercise 3

Write a function that takes a string as an argument and returns the number of capital letters in the string.

Hint: \texttt{'foo'.upper()} returns \texttt{'FOO'}.

9.4 Exercise 4

Write a function that takes two sequences \texttt{seq_a} and \texttt{seq_b} as arguments and returns True if every element in \texttt{seq_a} is also an element of \texttt{seq_b}, else False.

\begin{itemize}
  \item By “sequence” we mean a list, a tuple or a string.
  \item Do the exercise without using \texttt{sets} and set methods.
\end{itemize}
9.5 Exercise 5

When we cover the numerical libraries, we will see they include many alternatives for interpolation and function approximation.

Nevertheless, let’s write our own function approximation routine as an exercise.

In particular, without using any imports, write a function `linapprox` that takes as arguments

- A function \( f \) mapping some interval \([a,b]\) into \( \mathbb{R} \).
- Two scalars \( a \) and \( b \) providing the limits of this interval.
- An integer \( n \) determining the number of grid points.
- A number \( x \) satisfying \( a \leq x \leq b \).

and returns the piecewise linear interpolation of \( f \) at \( x \), based on \( n \) evenly spaced grid points \( a = \text{point}[0] < \text{point}[1] < \ldots < \text{point}[n-1] = b \).

Aim for clarity, not efficiency.

9.6 Exercise 6

Using list comprehension syntax, we can simplify the loop in the following code.

```python
In [63]: import numpy as np

n = 100
c_values = []
for i in range(n):
    e = np.random.randn()
    c_values.append(e)
```

10 Solutions

10.1 Exercise 1

10.1.1 Part 1 Solution:

Here’s one possible solution

```python
In [64]: x_vals = [1, 2, 3]
y_vals = [1, 1, 1]
sum([x * y for x, y in zip(x_vals, y_vals)])
```

Out[64]: 6

This also works

```python
In [65]: sum(x * y for x, y in zip(x_vals, y_vals))
```

Out[65]: 6
10.1.2 Part 2 Solution:

One solution is

In [66]: \text{sum}([x \% 2 == 0 \text{ for } x \text{ in range}(100)])

Out[66]: 50

This also works:

In [67]: \text{sum}(x \% 2 == 0 \text{ for } x \text{ in range}(100))

Out[67]: 50

Some less natural alternatives that nonetheless help to illustrate the flexibility of list comprehensions are

In [68]: \text{len}([x \text{ for } x \text{ in range}(100) \text{ if } x \% 2 == 0])

Out[68]: 50

and

In [69]: \text{sum}([1 \text{ for } x \text{ in range}(100) \text{ if } x \% 2 == 0])

Out[69]: 50

10.1.3 Part 3 Solution

Here’s one possibility

In [70]: pairs = ((2, 5), (4, 2), (9, 8), (12, 10))
   \text{sum}([x \% 2 == 0 \text{ and } y \% 2 == 0 \text{ for } x, y \text{ in pairs}])

Out[70]: 2

10.2 Exercise 2

In [71]: \text{def } p(x, \text{ coeff}): \text{ return sum}(a * x**i \text{ for } i, a \text{ in enumerate(coeff)})

In [72]: p(1, (2, 4))

Out[72]: 6
10.3 Exercise 3

Here’s one solution:

```python
In [73]: def f(string):
    count = 0
    for letter in string:
        if letter == letter.upper() and letter.isalpha():
            count += 1
    return count

f('The Rain in Spain')
```

```
Out[73]: 3
```

An alternative, more pythonic solution:

```python
In [74]: def count_uppercase_chars(s):
    return sum([c.isupper() for c in s])

count_uppercase_chars('The Rain in Spain')
```

```
Out[74]: 3
```

10.4 Exercise 4

Here’s a solution:

```python
In [75]: def f(seq_a, seq_b):
    is_subset = True
    for a in seq_a:
        if a not in seq_b:
            is_subset = False
    return is_subset

# == test == #
print(f([1, 2], [1, 2, 3]))
print(f([1, 2, 3], [1, 2]))
```

```
True
False
```

Of course, if we use the `sets` data type then the solution is easier

```python
In [76]: def f(seq_a, seq_b):
    return set(seq_a).issubset(set(seq_b))
```

```
```
10.5 Exercise 5

In [77]: def linapprox(f, a, b, n, x):
    
    Evaluates the piecewise linear interpolant of \( f \) at \( x \) on the interval \([a, b]\), with \( n \) evenly spaced grid points.

    Parameters
    ===========
    f : function
        The function to approximate
    x, a, b : scalars (floats or integers)
        Evaluation point and endpoints, with \( a \leq x \leq b \)
    n : integer
        Number of grid points

    Returns
    =======
    A float. The interpolant evaluated at \( x \)
    
    length_of_interval = b - a
    num_subintervals = n - 1
    step = length_of_interval / num_subintervals

    # === find first grid point larger than x === #
    point = a
    while point <= x:
        point += step

    # === x must lie between the gridpoints (point - step) and point === #
    u, v = point - step, point

    return f(u) + (x - u) * (f(v) - f(u)) / (v - u)

10.6 Exercise 6

Here's one solution.

In [78]: n = 100
    ϵ_values = [np.random.randn() for i in range(n)]