Python Basics

for Data Analysis

By Armin Khayati

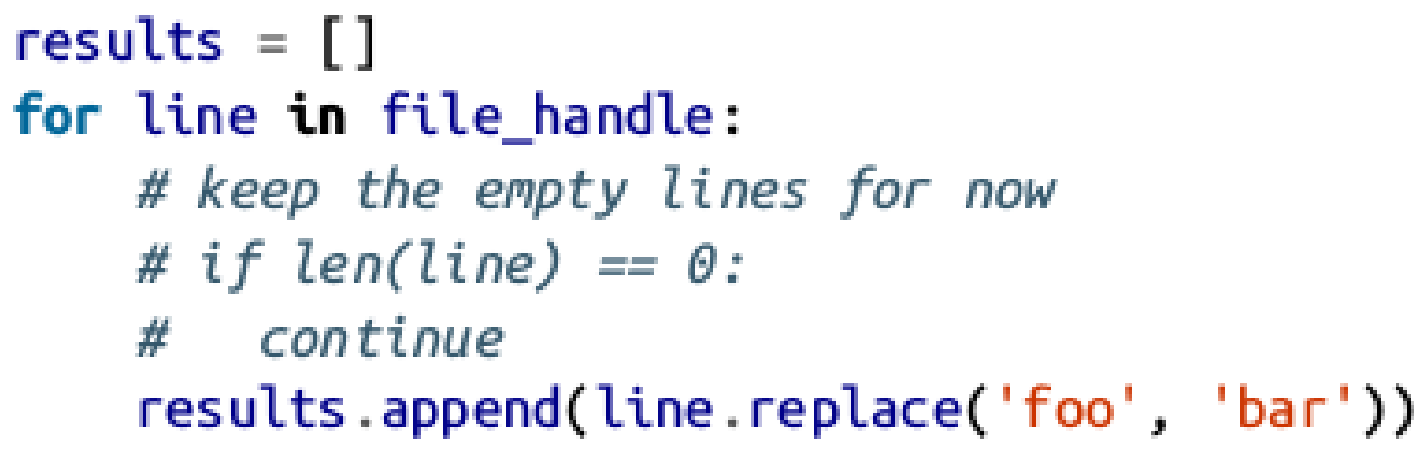
Python Basics : Everything is an Object

- Everything is an object (number, string, data structure, function, class, module) which is referred to as a Python object

- Each object has an associated type and internal data.

Python Basics : Comments

- Any text preceded by the hash mark (pound sign) # is ignored by the Python interpreter.



```python
results = []
for line in file_handle:
    # keep the empty lines for now
    # if len(line) == 0:
    #   continue
    results.append(line.replace('foo', 'bar'))
```

Python Basics : Function and object method calls

- You call functions using parentheses and passing zero or more arguments, optionally assigning the returned value to a variable

```
result = f(x, y, z)
g()
```

- Almost every object in Python has attached functions, known as methods

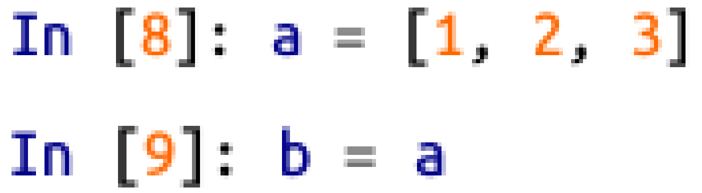
```
obj.some_method(x, y, z)
```

- Functions can take both positional and keyword arguments

```
result = f(a, b, c, d=5, e='foo')
```

Python Basics : Variables and argument passing

- When assigning a variable (or name) in Python, you are creating a reference to the object on the right-hand side of the equals sign.



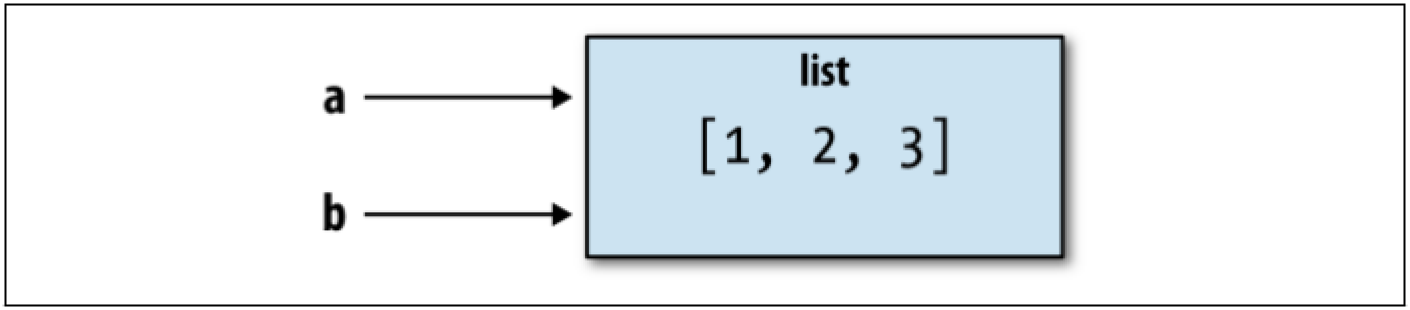
```
In [8]: a = [1, 2, 3]

In [9]: b = a
```

- In Python, a and b actually now refer to the same object, the original list [1, 2, 3]

Python Basics : Variables and argument passing

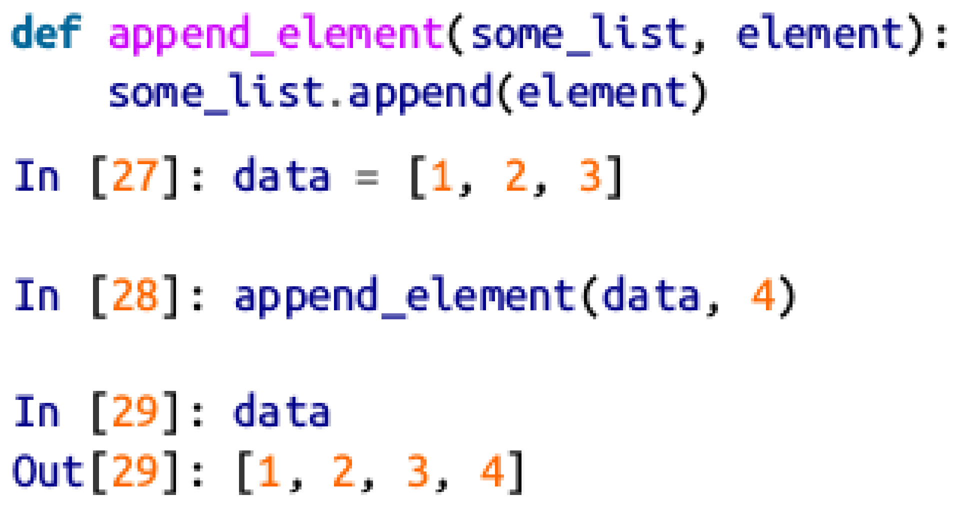
- You can prove this to yourself by appending an element to a and then examining b

```
In [10]: a.append(4)

In [11]: b
Out[11]: [1, 2, 3, 4]
```



Python Basics : Variables and argument passing

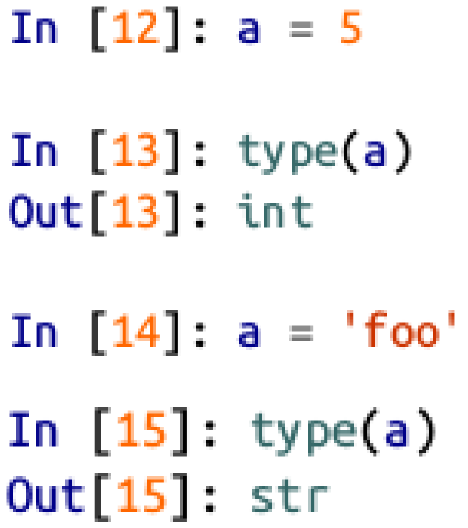
- When you pass objects as arguments to a function, new local variables are created referencing the original objects without any copying.
- If you bind a new object to a variable inside a function, that change will not be reflected in the parent scope.



```
def append_element(some_list, element):
    some_list.append(element)

In [27]: data = [1, 2, 3]

In [28]: append_element(data, 4)

In [29]: data
Out[29]: [1, 2, 3, 4]
```

Python Basics : Dynamic references, strong types

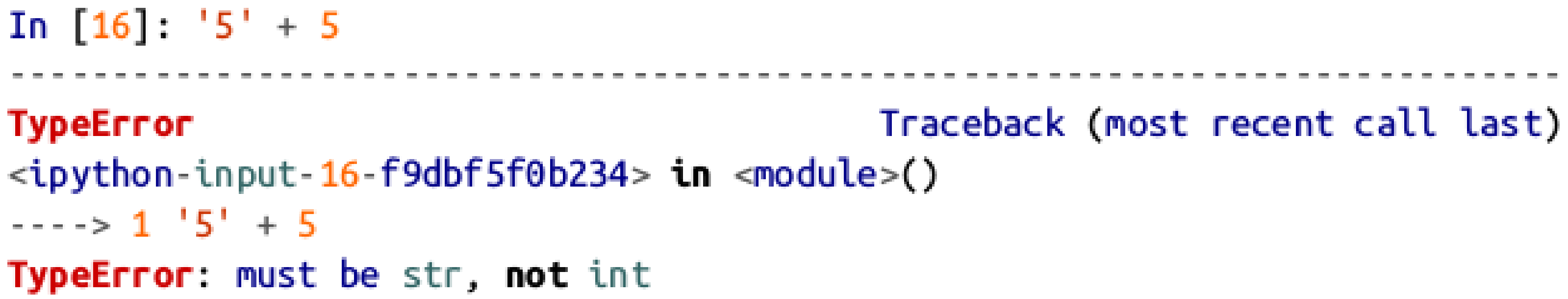
- In contrast with many compiled languages, such as Java and C++, object references in Python have no type associated with them.



```
In [12]: a = 5

In [13]: type(a)
Out[13]: int

In [14]: a = 'foo'

In [15]: type(a)
Out[15]: str
```

Python Basics : Dynamic references, strong types

- Variables are names for objects within a particular name-space; the type information is stored in the object itself
- Python is a "typed language" ; consider this example

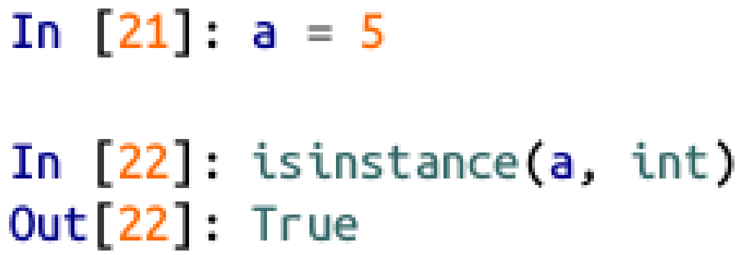


```
In [16]: '5' + 5
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-16-f9dbf5f0b234> in <module>()
----> 1 '5' + 5
TypeError: must be str, not int
```

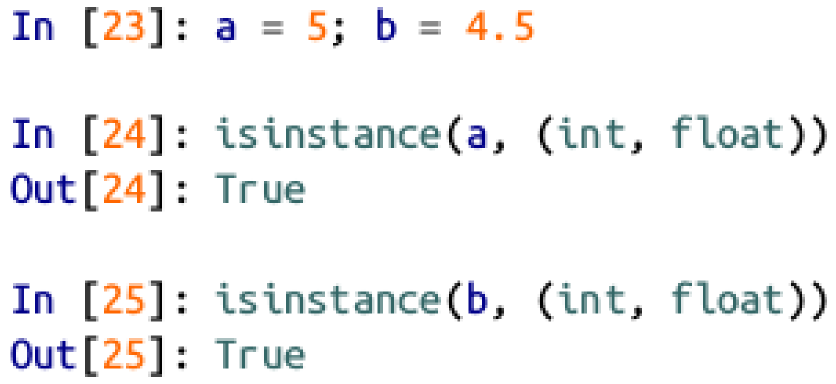
- Python is considered a strongly typed language

Python Basics : Dynamic references, strong types

- You can check that an object is an instance of a particular type using the isinstance function
- `isinstance` can accept a tuple of types if you want to check that an object's type is among those present in the tuple



```
In [21]: a = 5

In [22]: isinstance(a, int)
Out[22]: True
```



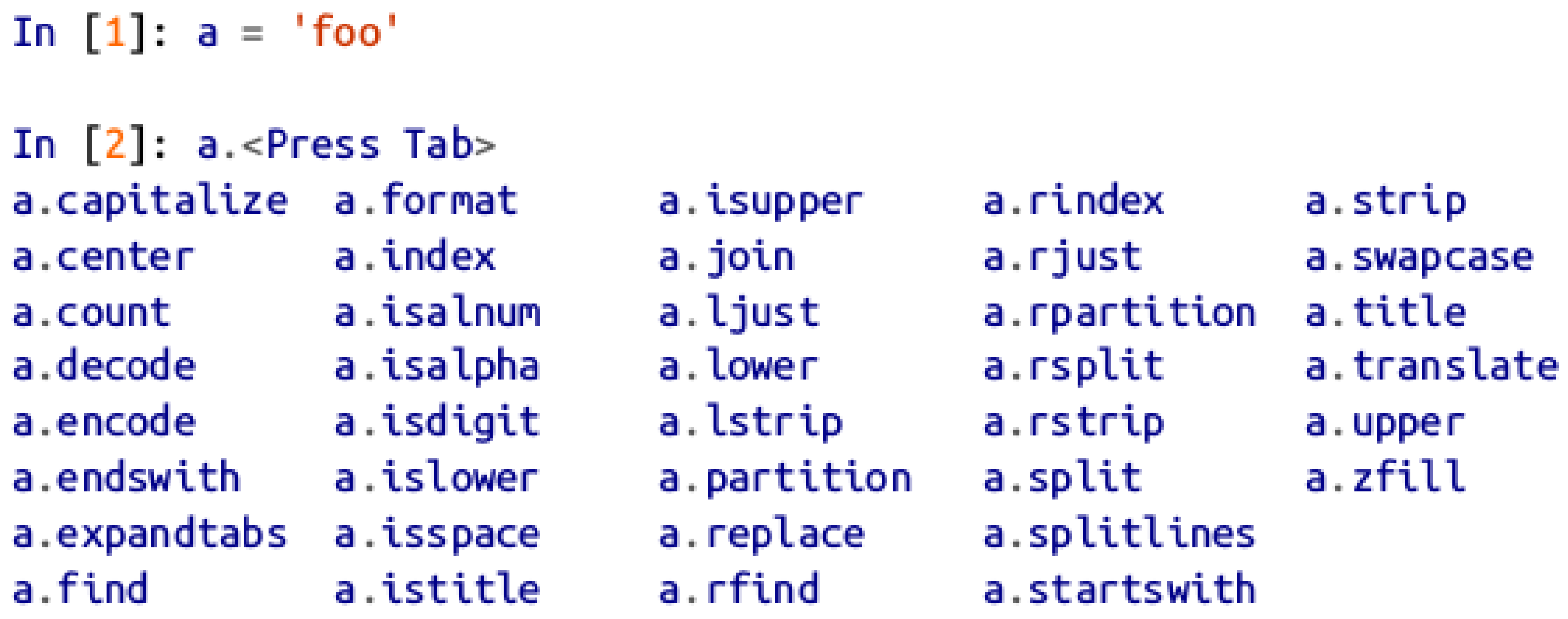
```
In [23]: a = 5; b = 4.5

In [24]: isinstance(a, (int, float))
Out[24]: True

In [25]: isinstance(b, (int, float))
Out[25]: True
```

Python Basics : Attributes and methods

- Objects in Python have attributes (other Python objects stored “inside”the object)
- Objects in Python methods (functions associated with an object that can have access to the object’s internal data)

Python Basics : Attributes and methods

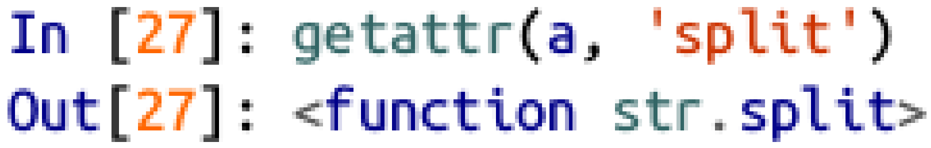
- Both of them are accessed via the syntax `obj.attribute_name`



```
In [1]: a = 'foo'

In [2]: a.<Press Tab>
a.capitalize  a.format      a.isupper     a.rindex      a.strip
a.center      a.index       a.join        a.rjust       a.swapcase
a.count       a.isalnum     a.ljust       a.rpartition  a.title
a.decode      a.isalpha     a.lower       a.rsplit      a.translate
a.encode      a.isdigit     a.lstrip      a.rstrip      a.upper
a.endswith    a.islower     a.partition   a.split       a.zfill
a.expandtabs  a.isspace     a.replace     a.splitlines
a.find        a.istitle     a.rfind       a.startswith
```

Python Basics : Attributes and methods

- Attributes and methods can also be accessed by name via the `getattr` function



```
In [27]: getattr(a, 'split')
Out[27]: <function str.split>
```

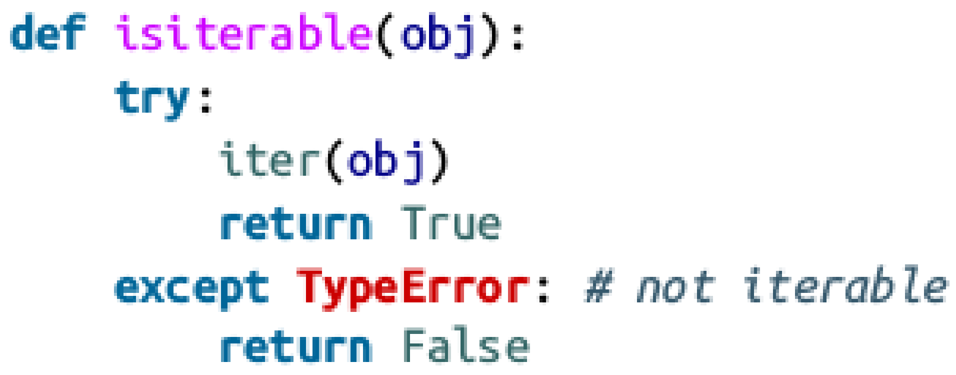
Python Basics : Duck typing

- Often you may not care about the type of an object but rather only whether it has certain methods or behavior.

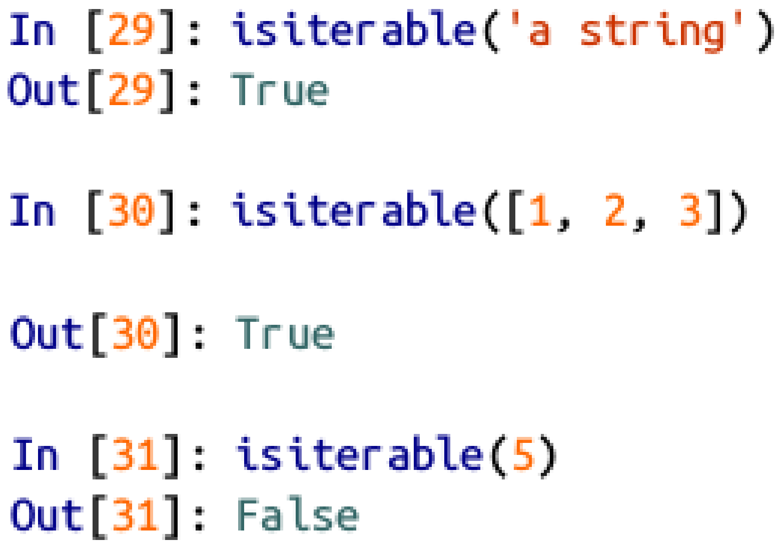
- This is sometimes called “duck typing,” after the saying “If it walks like a duck and quacks like a duck, then it’s a duck.”

Python Basics : Duck typing

- For example, you can verify that an object is iterable if it implemented the iterator protocol. For many objects, this means it has a `__iter__` “magic method” .



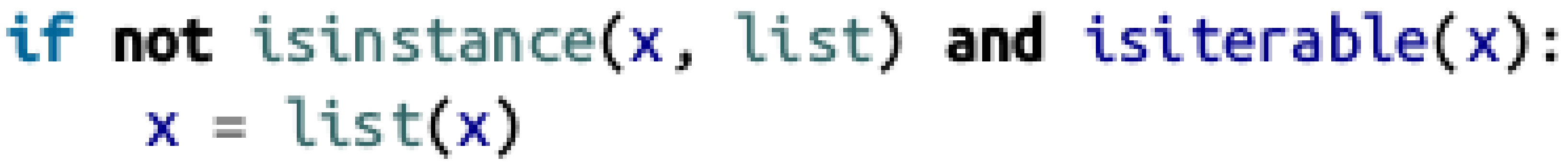
```python
def isiterable(obj):
    try:
        iter(obj)
        return True
    except TypeError: # not iterable
        return False
```



```
In [29]: isiterable('a string')
Out[29]: True

In [30]: isiterable([1, 2, 3])

Out[30]: True

In [31]: isiterable(5)
Out[31]: False
```

Python Basics : Duck typing

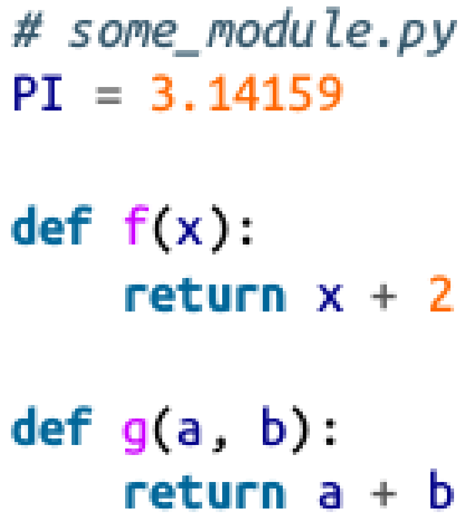
- Write functions that can accept multiple kinds of input.
- A function that can accept any kind of sequence (list, tuple, ndarray) or even an iterator. You can first check if the object is a list (or a NumPy array) and, if it is not, convert it to be one.



```python
if not isinstance(x, list) and isiterable(x):
    x = list(x)
```

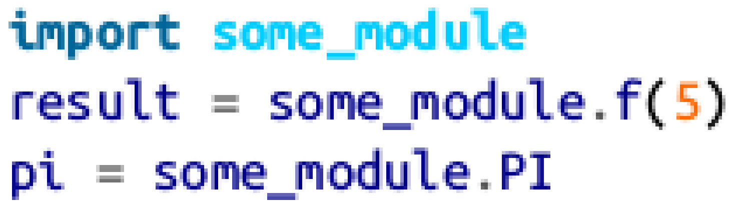
Python Basics : Imports

- In Python a module is simply a file with the `.py` extension containing Python code. Suppose that we had the following module:



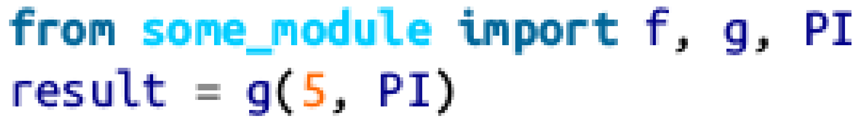
```
# some_module.py
PI = 3.14159

def f(x):
    return x + 2

def g(a, b):
    return a + b
```

- If we wanted to access the variables and functions defined in some_module.py, from another file in the same directory we could do



```
import some_module
result = some_module.f(5)
pi = some_module.PI
```

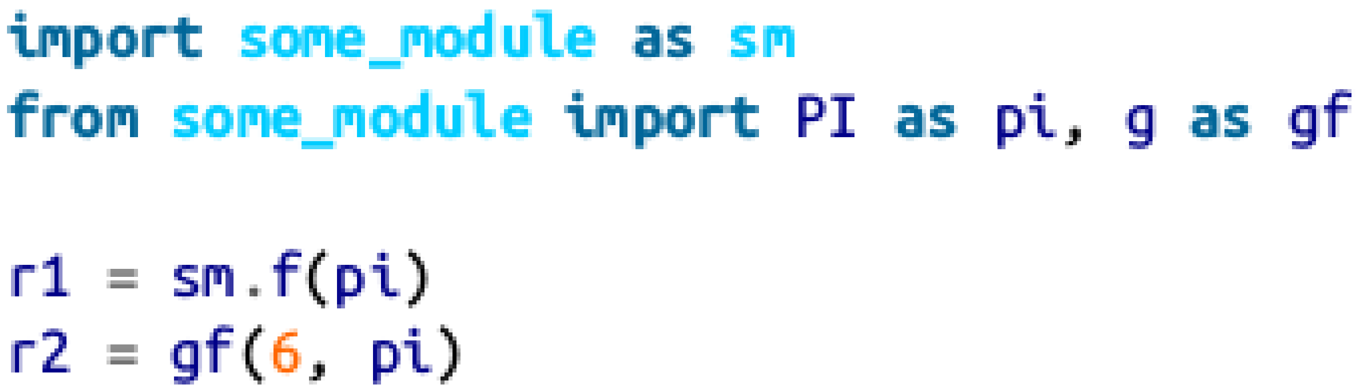
OR



```
from some_module import f, g, PI
result = g(5, PI)
```

Python Basics : Imports

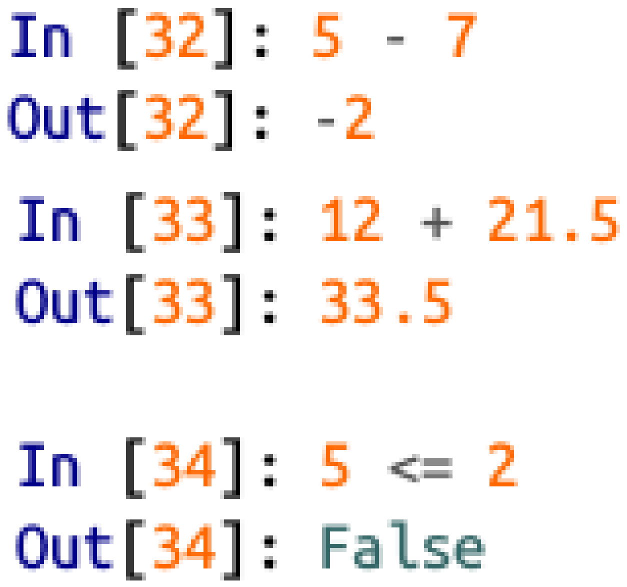
- By using the as keyword you can give imports different variable names.



```python
import some_module as sm
from some_module import PI as pi, g as gf

r1 = sm.f(pi)
r2 = gf(6, pi)
```

Python Basics : Binary operators and comparisons

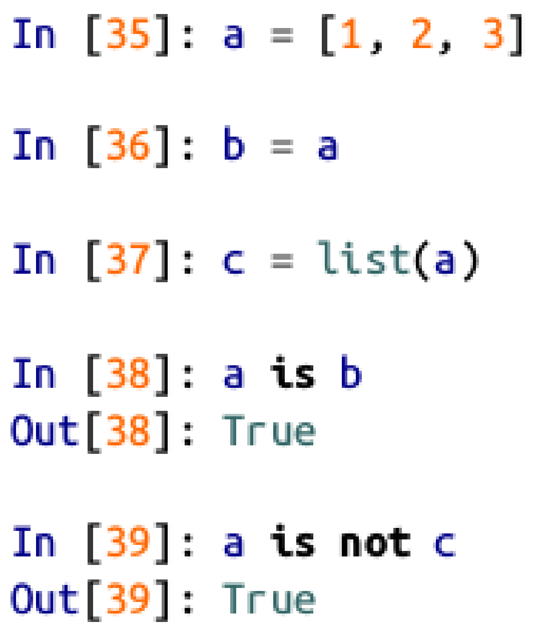
- Most of the binary math operations and comparisons are as you might expect



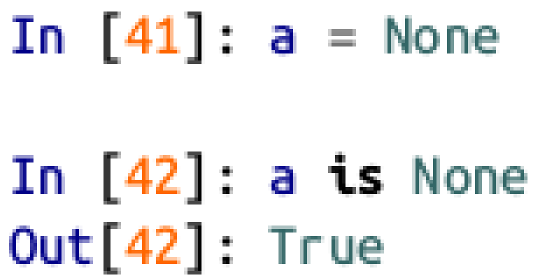
```
In [32]: 5 - 7
Out[32]: -2

In [33]: 12 + 21.5
Out[33]: 33.5

In [34]: 5 <= 2
Out[34]: False
```

Python Basics : Binary operators and comparisons

- To check if two references refer to the same object, use the `is` keyword.
- `is not` is also perfectly valid if you want to check that two objects are not the same.



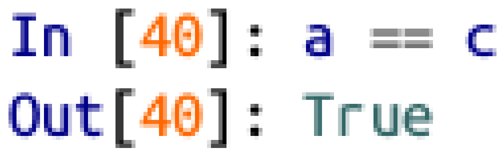
```
In [35]: a = [1, 2, 3]

In [36]: b = a

In [37]: c = list(a)

In [38]: a is b
Out[38]: True

In [39]: a is not c
Out[39]: True
```



```
In [41]: a = None

In [42]: a is None
Out[42]: True
```

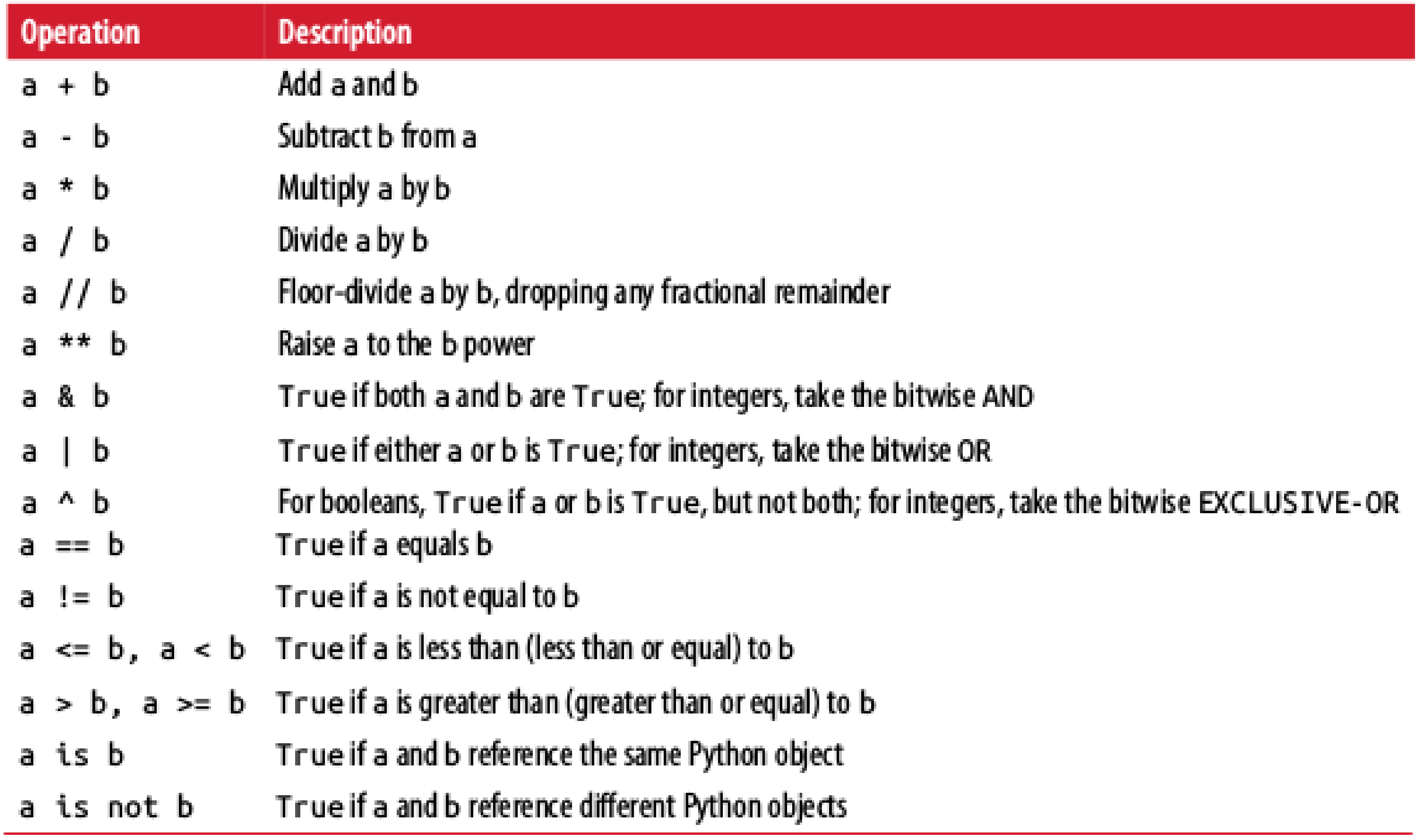
Python Basics : Binary operators and comparisons

- `list` always creates a new Python list (i.e., a copy), we can be sure that c is distinct from a .
- Comparing with '`is`' is not the same as the == operator.



```
In [40]: a == c
Out[40]: True
```

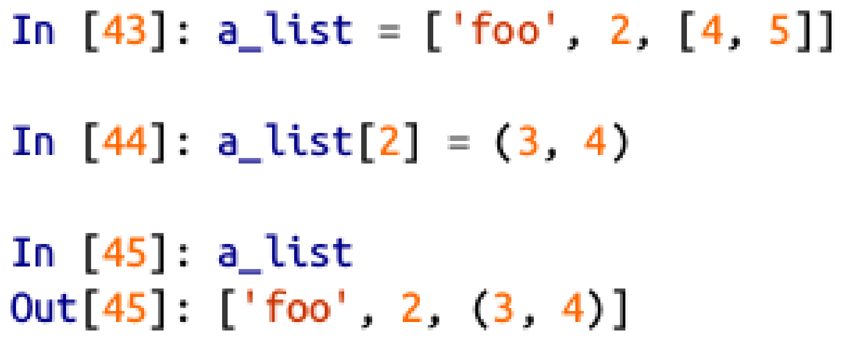
Python Basics : Binary operators and comparisons



Operation	Description
a + b	Add a and b
a - b	Subtract b from a
a * b	Multiply a by b
a / b	Divide a by b
a // b	Floor-divide a by b, dropping any fractional remainder
a ** b	Raise a to the b power
a & b	True if both a and b are True; for integers, take the bitwise AND
a | b	True if either a or b is True; for integers, take the bitwise OR
a ^ b	For booleans, True if a or b is True, but not both; for integers, take the bitwise EXCLUSIVE-OR
a == b	True if a equals b
a != b	True if a is not equal to b
a <= b, a < b	True if a is less than (less than or equal) to b
a > b, a >= b	True if a is greater than (greater than or equal) to b
a is b	True if a and b reference the same Python object
a is not b	True if a and b reference different Python objects

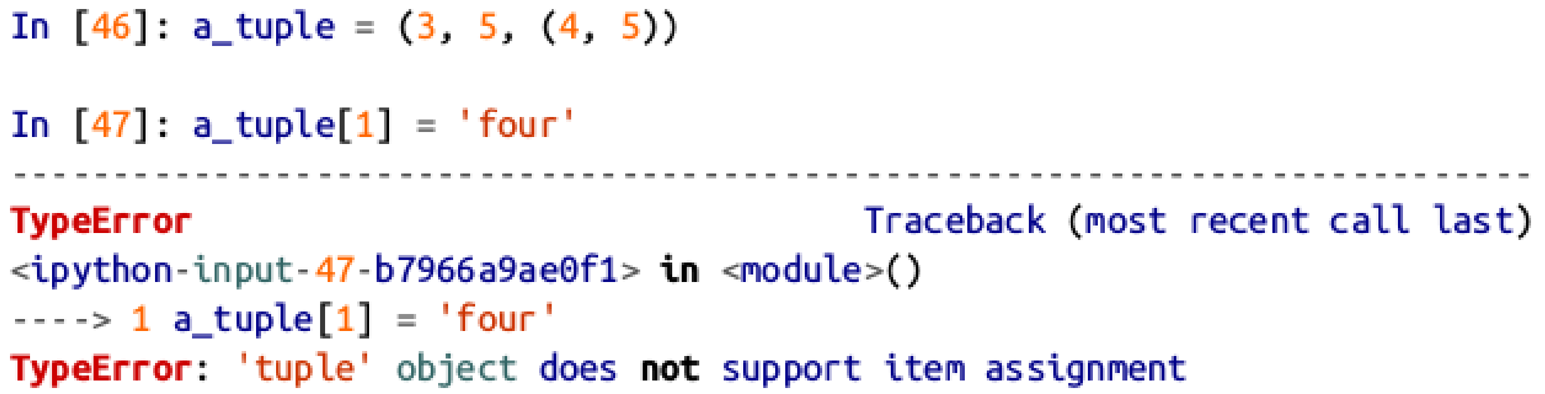
Python Basics : Mutable and immutable objects

- Most objects in Python, such as lists, dicts, NumPy arrays, and most user-defined types (classes), are mutable.



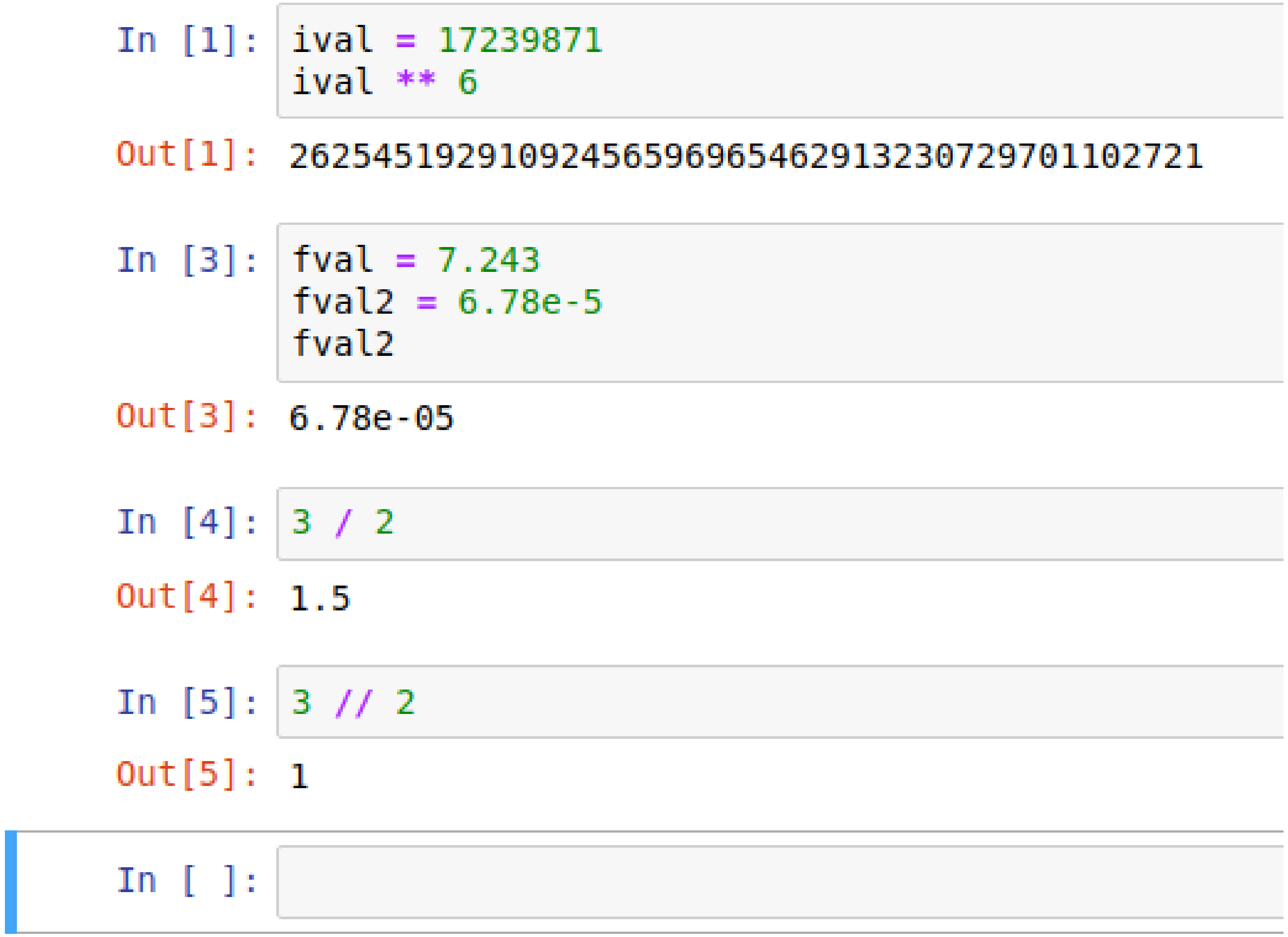
```
In [43]: a_list = ['foo', 2, [4, 5]]

In [44]: a_list[2] = (3, 4)

In [45]: a_list
Out[45]: ['foo', 2, (3, 4)]
```

- Others, like strings and tuples, are immutable.



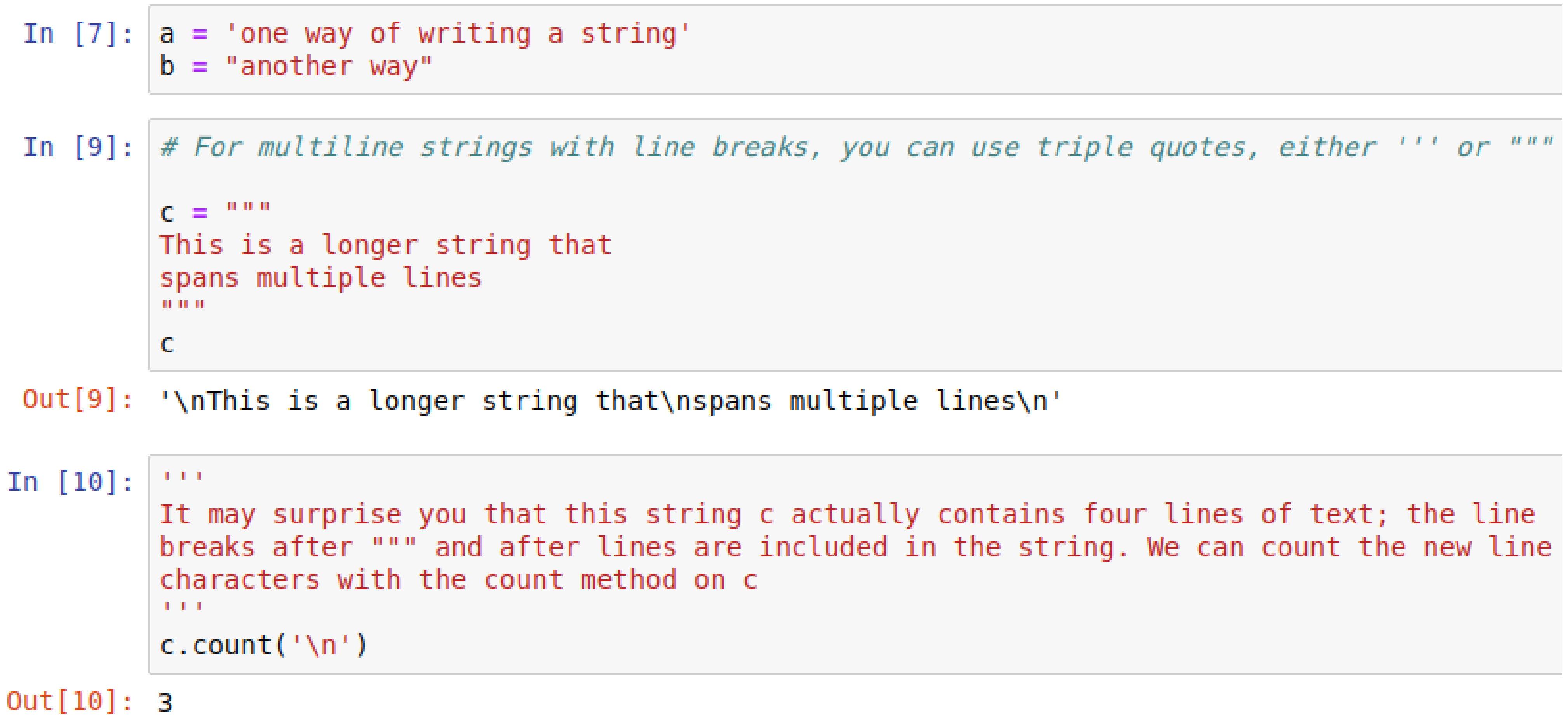
```
In [46]: a_tuple = (3, 5, (4, 5))

In [47]: a_tuple[1] = 'four'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-47-b7966a9ae0f1> in <module>()
----> 1 a_tuple[1] = 'four'
TypeError: 'tuple' object does not support item assignment
```

Python Basics : Numeric Types



```
In [1]: ival = 17239871
        ival ** 6

Out[1]: 26254519291092456596965462913230729701102721

In [3]: fval = 7.243
        fval2 = 6.78e-5
        fval2

Out[3]: 6.78e-05

In [4]: 3 / 2

Out[4]: 1.5

In [5]: 3 // 2

Out[5]: 1

In [ ]:
```

Python Basics : Strings

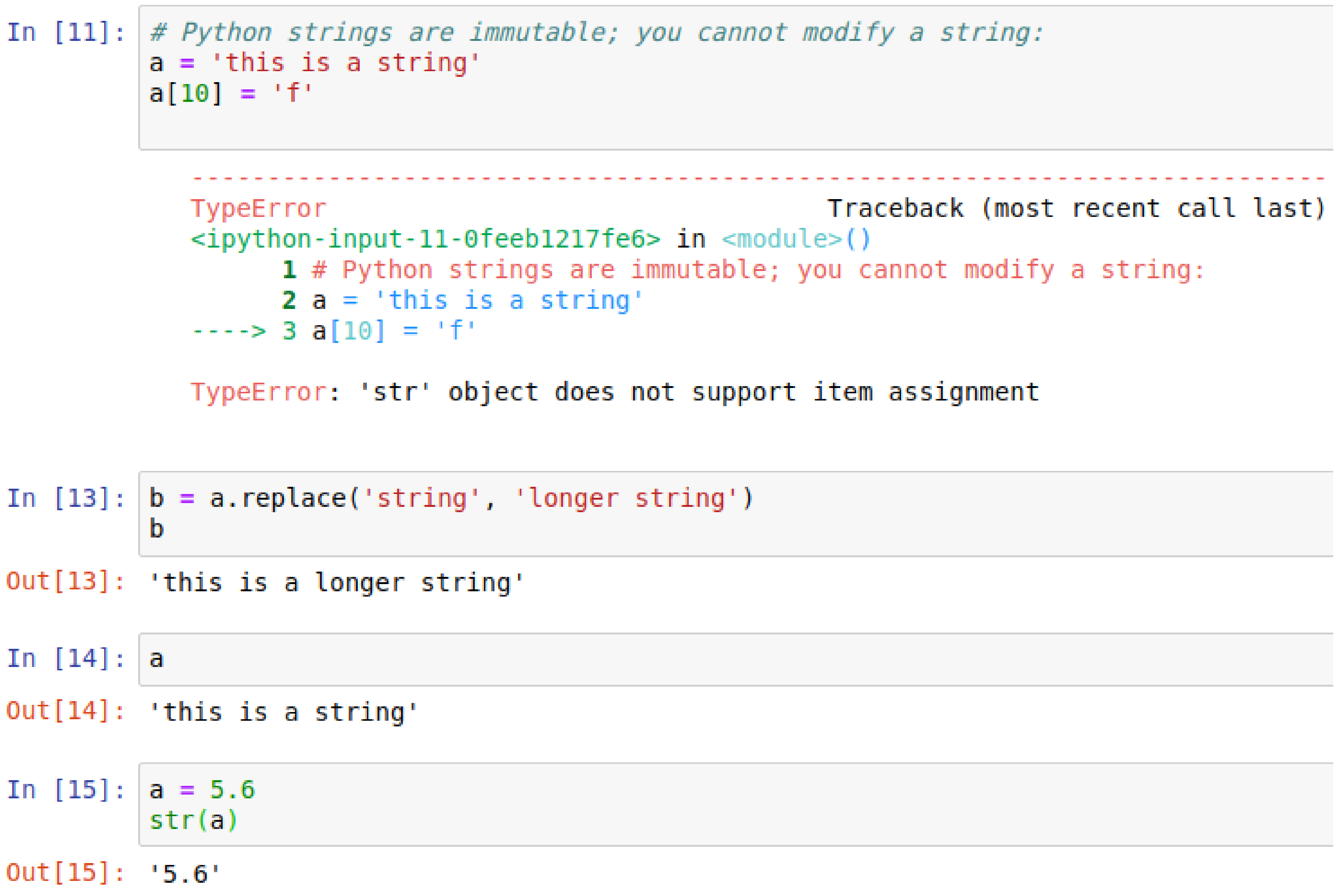


```
In [7]:  a = 'one way of writing a string'
         b = "another way"
```

```
In [9]:  # For multiline strings with line breaks, you can use triple quotes, either ''' or """

         c = """
         This is a longer string that
         spans multiple lines
         """
         c

Out[9]:  '\nThis is a longer string that\nspans multiple lines\n'
```

```
In [10]: '''
         It may surprise you that this string c actually contains four lines of text; the line
         breaks after """ and after lines are included in the string. We can count the new line
         characters with the count method on c
         '''
         c.count('\n')

Out[10]: 3
```

Python Basics : Strings



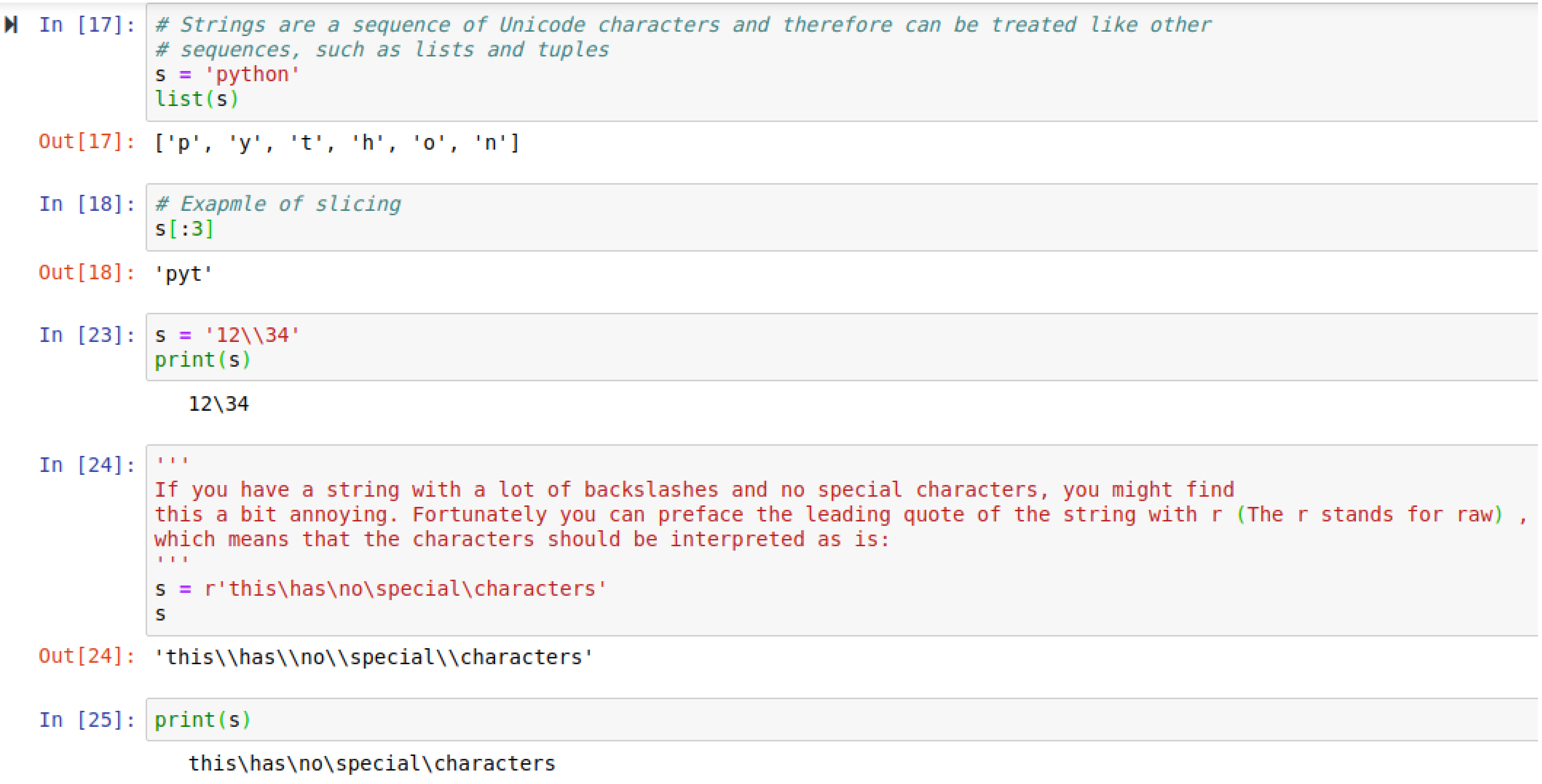
```
In [11]: # Python strings are immutable; you cannot modify a string:
         a = 'this is a string'
         a[10] = 'f'

         ---------------------------------------------------------------------------
         TypeError                                 Traceback (most recent call last)
         <ipython-input-11-0feeb1217fe6> in <module>()
               1 # Python strings are immutable; you cannot modify a string:
               2 a = 'this is a string'
         ----> 3 a[10] = 'f'

         TypeError: 'str' object does not support item assignment
```

```
In [13]: b = a.replace('string', 'longer string')
         b

Out[13]: 'this is a longer string'
```

```
In [14]: a

Out[14]: 'this is a string'
```

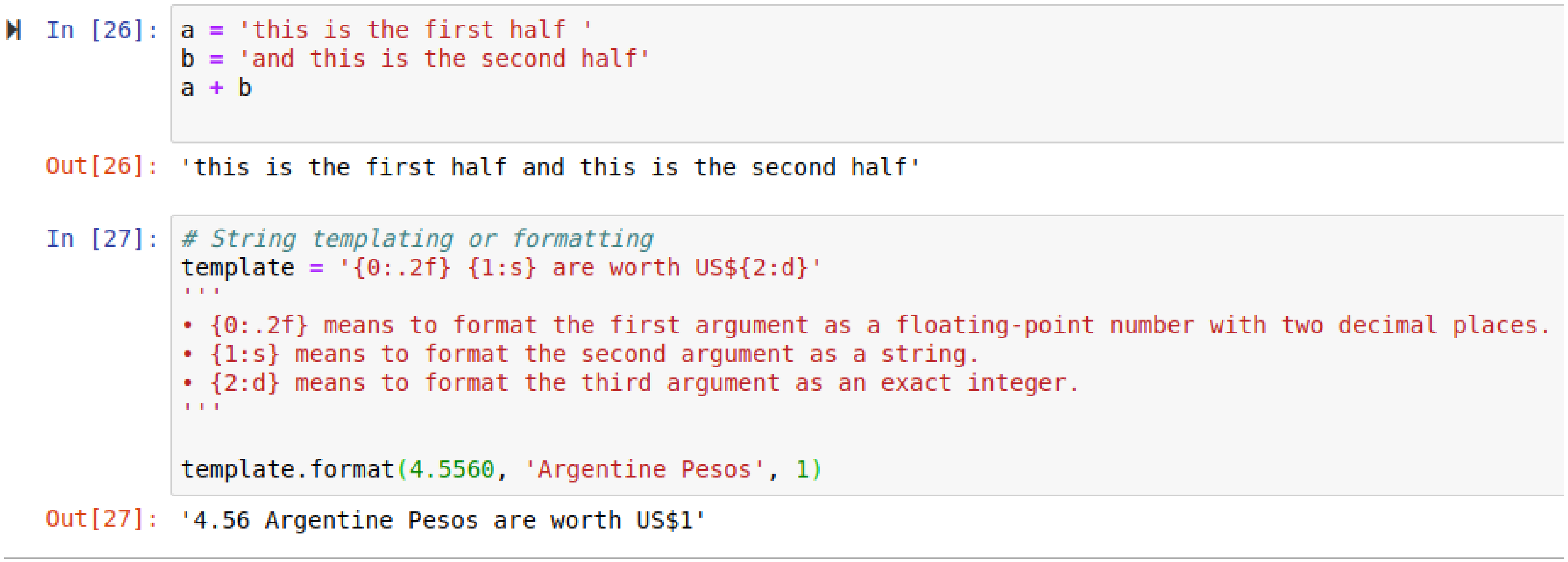
```
In [15]: a = 5.6
         str(a)

Out[15]: '5.6'
```

Python Basics : Strings



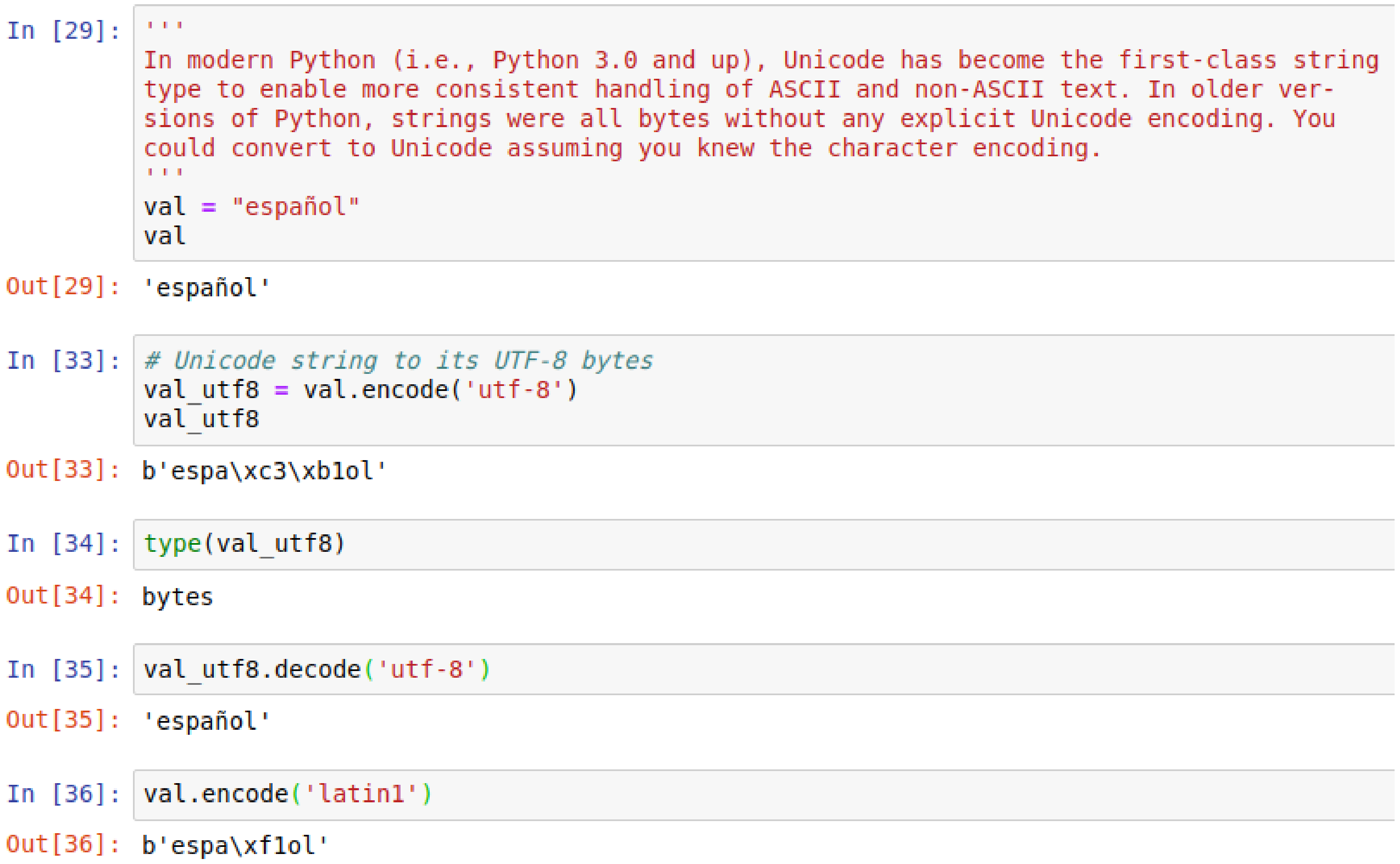
```
In [17]: # Strings are a sequence of Unicode characters and therefore can be treated like other
         # sequences, such as lists and tuples
         s = 'python'
         list(s)

Out[17]: ['p', 'y', 't', 'h', 'o', 'n']

In [18]: # Exapmle of slicing
         s[:3]

Out[18]: 'pyt'

In [23]: s = '12\\34'
         print(s)

           12\34

In [24]: '''
         If you have a string with a lot of backslashes and no special characters, you might find
         this a bit annoying. Fortunately you can preface the leading quote of the string with r (The r stands for raw) ,
         which means that the characters should be interpreted as is:
         '''
         s = r'this\has\no\special\characters'
         s

Out[24]: 'this\\has\\no\\special\\characters'

In [25]: print(s)

           this\has\no\special\characters
```

Python Basics : Strings



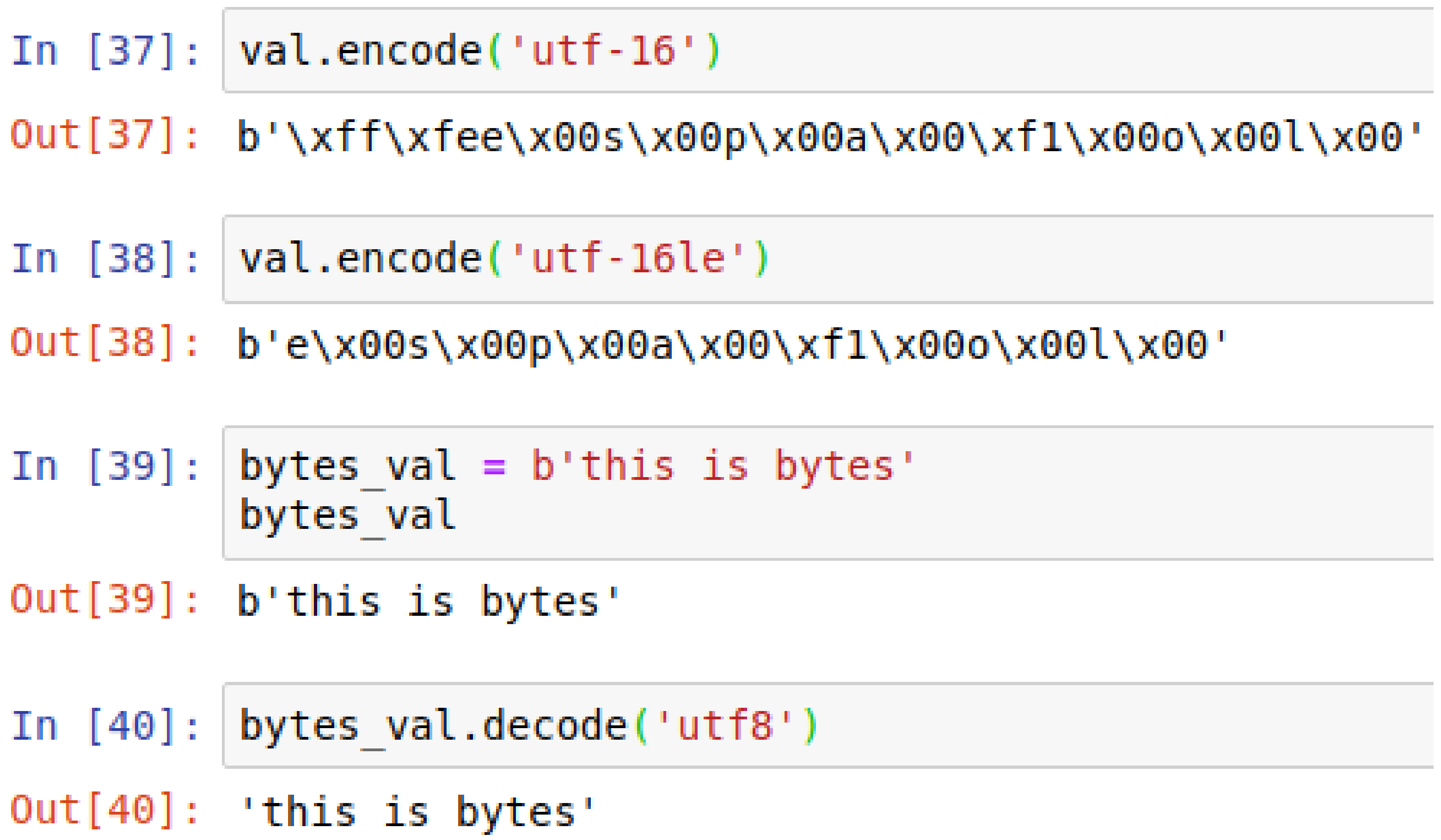
```
In [26]: a = 'this is the first half '
         b = 'and this is the second half'
         a + b

Out[26]: 'this is the first half and this is the second half'

In [27]: # String templating or formatting
         template = '{0:.2f} {1:s} are worth US${2:d}'
         '''
         • {0:.2f} means to format the first argument as a floating-point number with two decimal places.
         • {1:s} means to format the second argument as a string.
         • {2:d} means to format the third argument as an exact integer.
         '''

         template.format(4.5560, 'Argentine Pesos', 1)

Out[27]: '4.56 Argentine Pesos are worth US$1'
```

Python Basics :
Bytes and Unicode



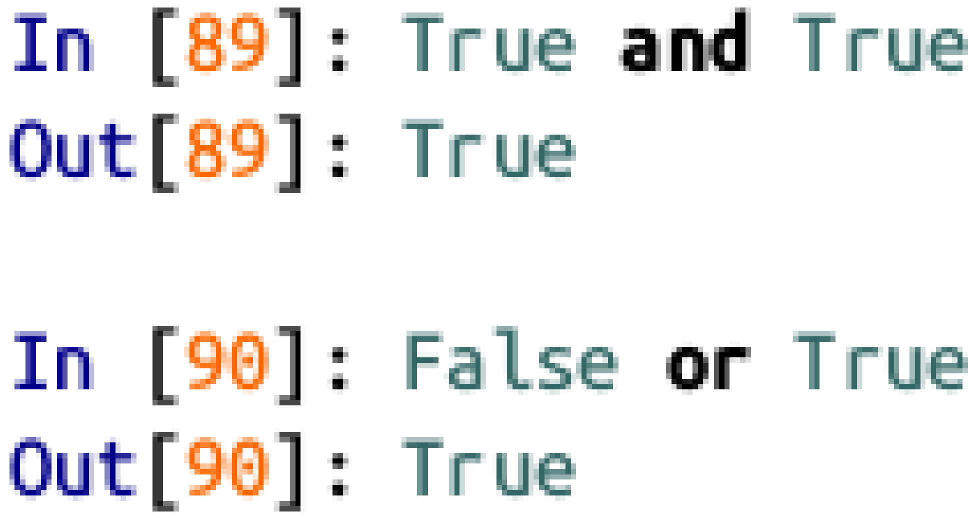
```
In [29]: '''
         In modern Python (i.e., Python 3.0 and up), Unicode has become the first-class string
         type to enable more consistent handling of ASCII and non-ASCII text. In older ver-
         sions of Python, strings were all bytes without any explicit Unicode encoding. You
         could convert to Unicode assuming you knew the character encoding.
         '''
         val = "español"
         val

Out[29]: 'español'

In [33]: # Unicode string to its UTF-8 bytes
         val_utf8 = val.encode('utf-8')
         val_utf8

Out[33]: b'espa\xc3\xb1ol'

In [34]: type(val_utf8)

Out[34]: bytes

In [35]: val_utf8.decode('utf-8')

Out[35]: 'español'

In [36]: val.encode('latin1')

Out[36]: b'espa\xf1ol'
```

Python Basics : Bytes and Unicode



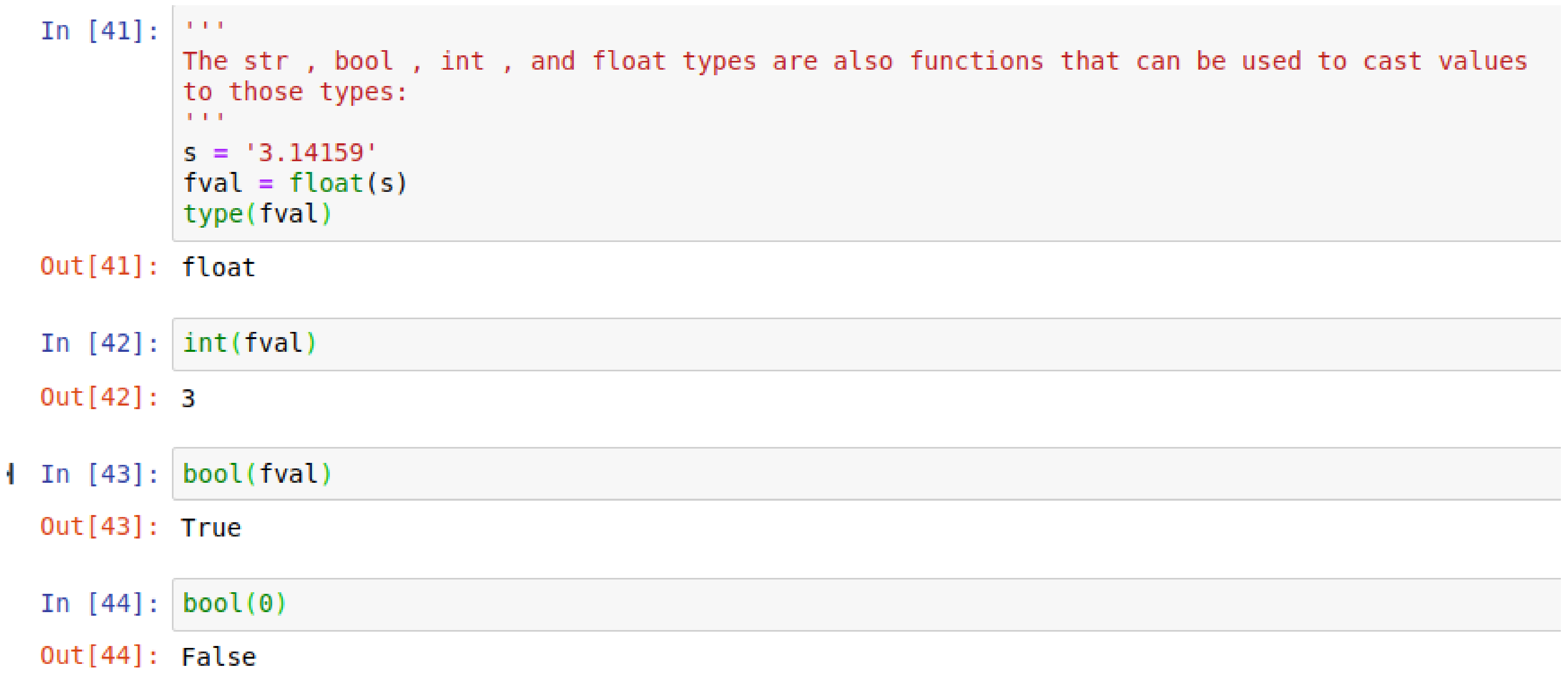
```
In [37]: val.encode('utf-16')

Out[37]: b'\xff\xfee\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'

In [38]: val.encode('utf-16le')

Out[38]: b'e\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'

In [39]: bytes_val = b'this is bytes'
         bytes_val

Out[39]: b'this is bytes'

In [40]: bytes_val.decode('utf8')

Out[40]: 'this is bytes'
```

Python Basics : Booleans



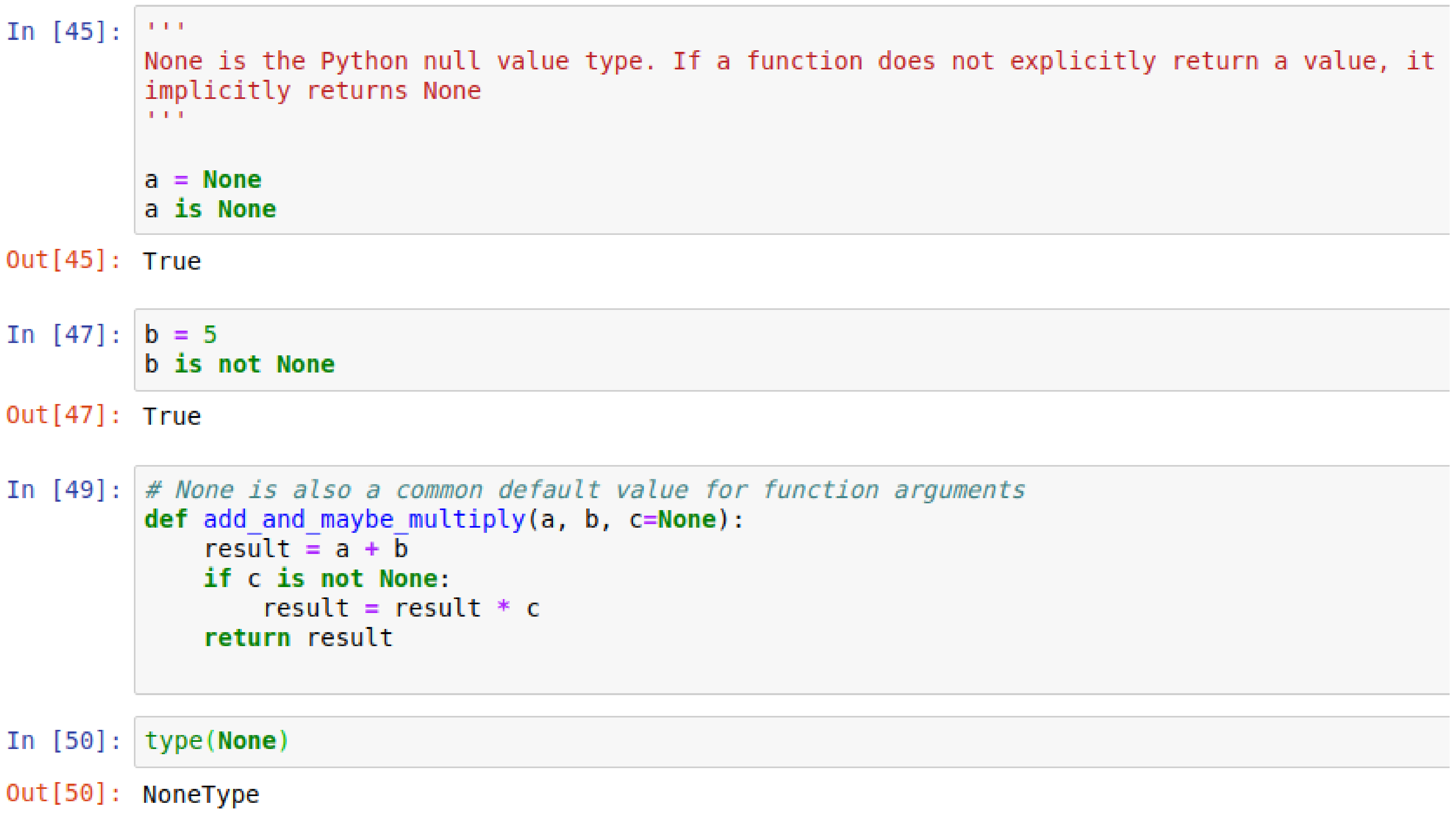
```
In [89]: True and True
Out[89]: True

In [90]: False or True
Out[90]: True
```

Python Basics : Type casting



```
In [41]: '''
         The str , bool , int , and float types are also functions that can be used to cast values
         to those types:
         '''
         s = '3.14159'
         fval = float(s)
         type(fval)
Out[41]: float

In [42]: int(fval)
Out[42]: 3

In [43]: bool(fval)
Out[43]: True

In [44]: bool(0)
Out[44]: False
```

Python Basics : None



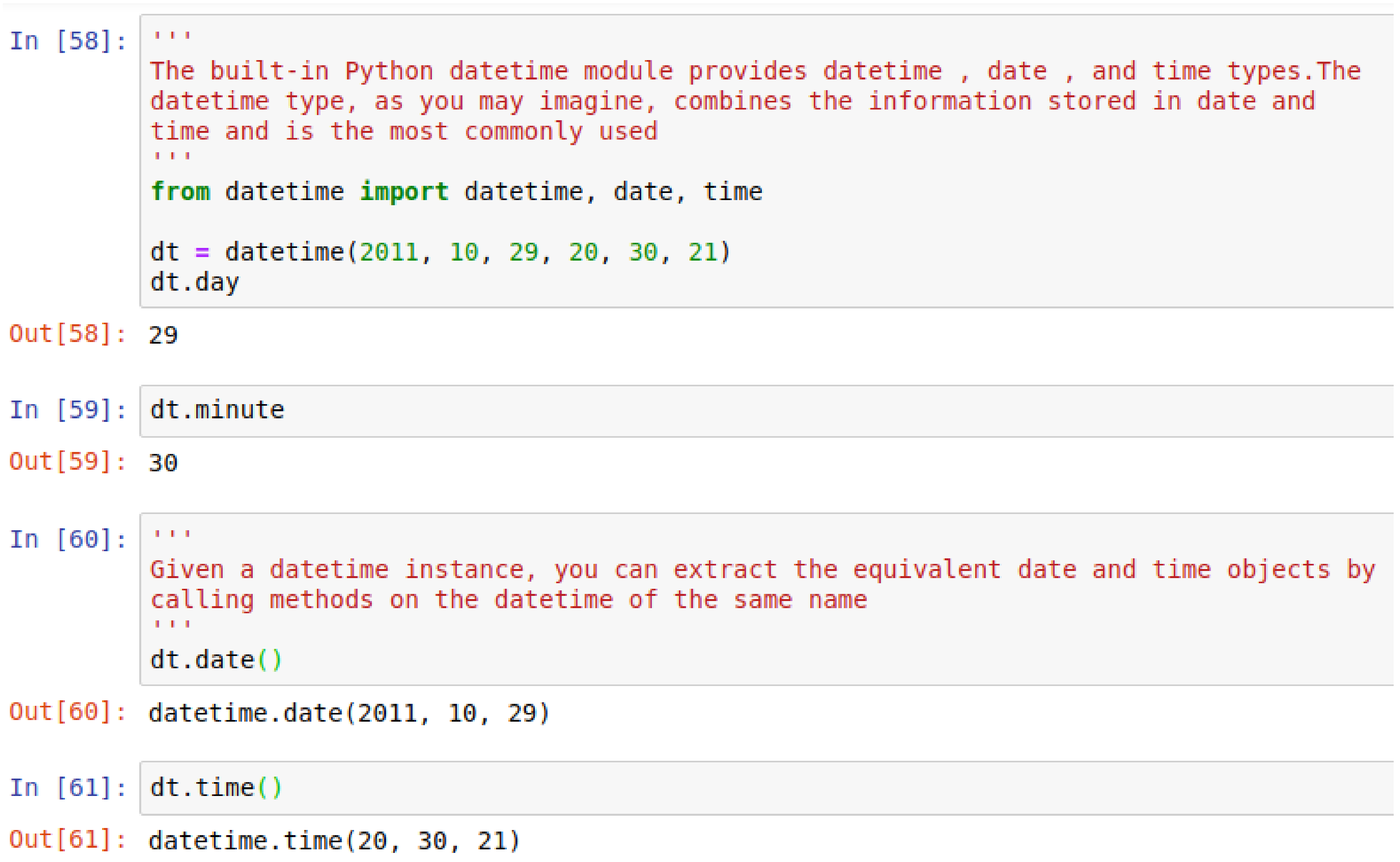
```
In [45]: '''
         None is the Python null value type. If a function does not explicitly return a value, it
         implicitly returns None
         '''

         a = None
         a is None

Out[45]: True
```

```
In [47]: b = 5
         b is not None

Out[47]: True
```

```
In [49]: # None is also a common default value for function arguments
         def add_and_maybe_multiply(a, b, c=None):
             result = a + b
             if c is not None:
                 result = result * c
             return result
```

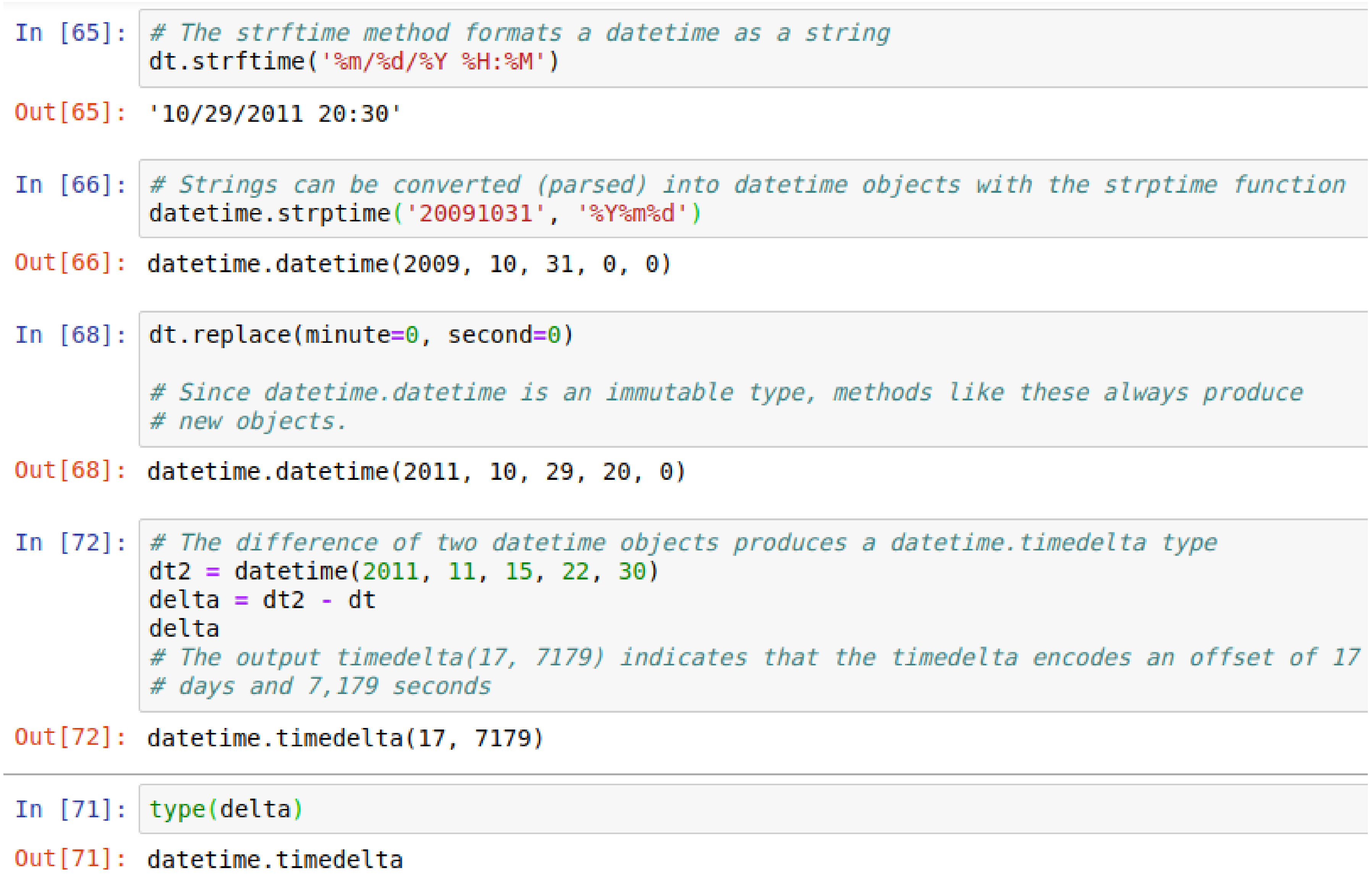
```
In [50]: type(None)

Out[50]: NoneType
```

Python Basics :
Dates and times



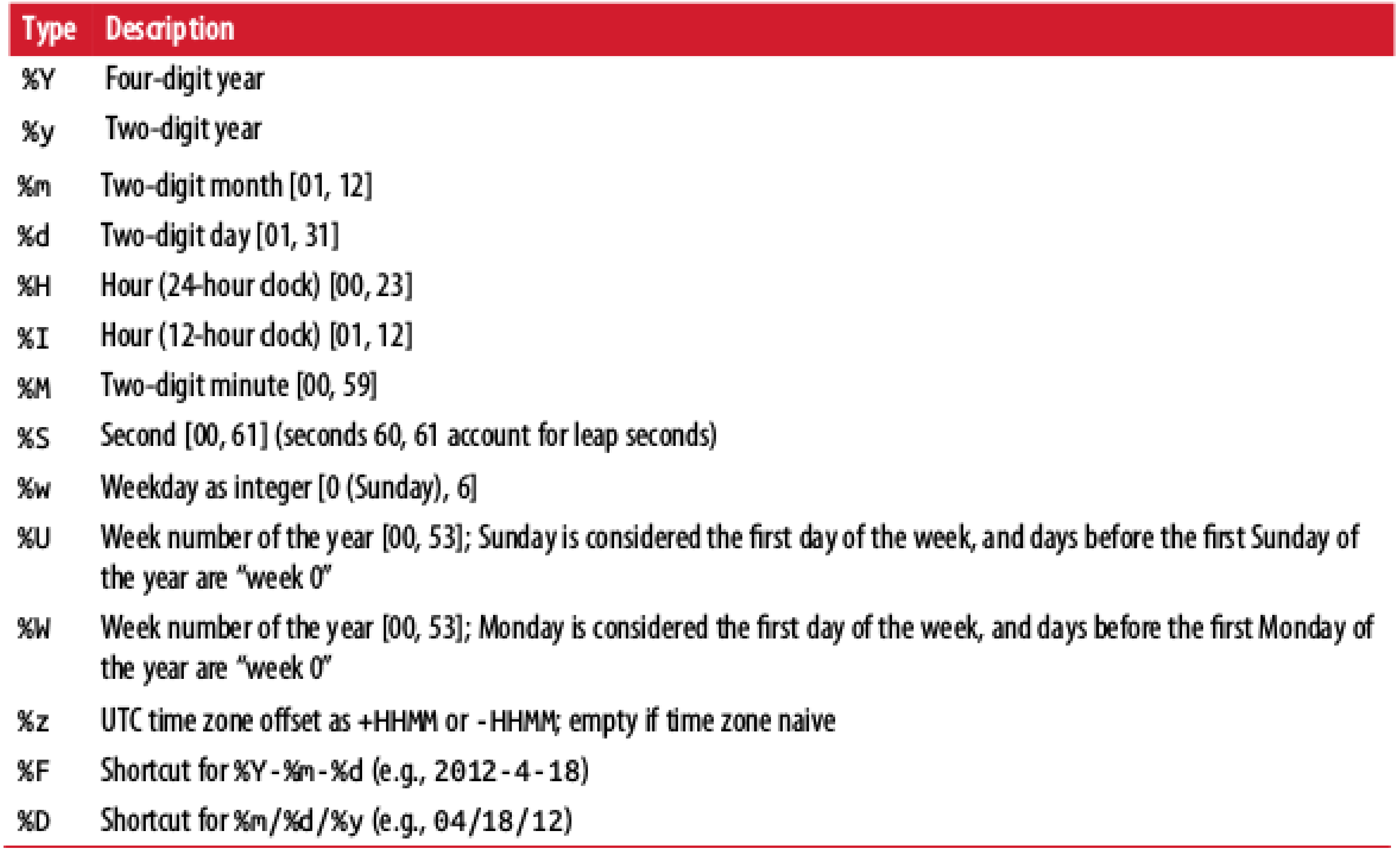
```
In [58]: '''
         The built-in Python datetime module provides datetime , date , and time types.The
         datetime type, as you may imagine, combines the information stored in date and
         time and is the most commonly used
         '''
         from datetime import datetime, date, time

         dt = datetime(2011, 10, 29, 20, 30, 21)
         dt.day

Out[58]: 29

In [59]: dt.minute

Out[59]: 30

In [60]: '''
         Given a datetime instance, you can extract the equivalent date and time objects by
         calling methods on the datetime of the same name
         '''
         dt.date()

Out[60]: datetime.date(2011, 10, 29)

In [61]: dt.time()

Out[61]: datetime.time(20, 30, 21)
```

Python Basics : Dates and times



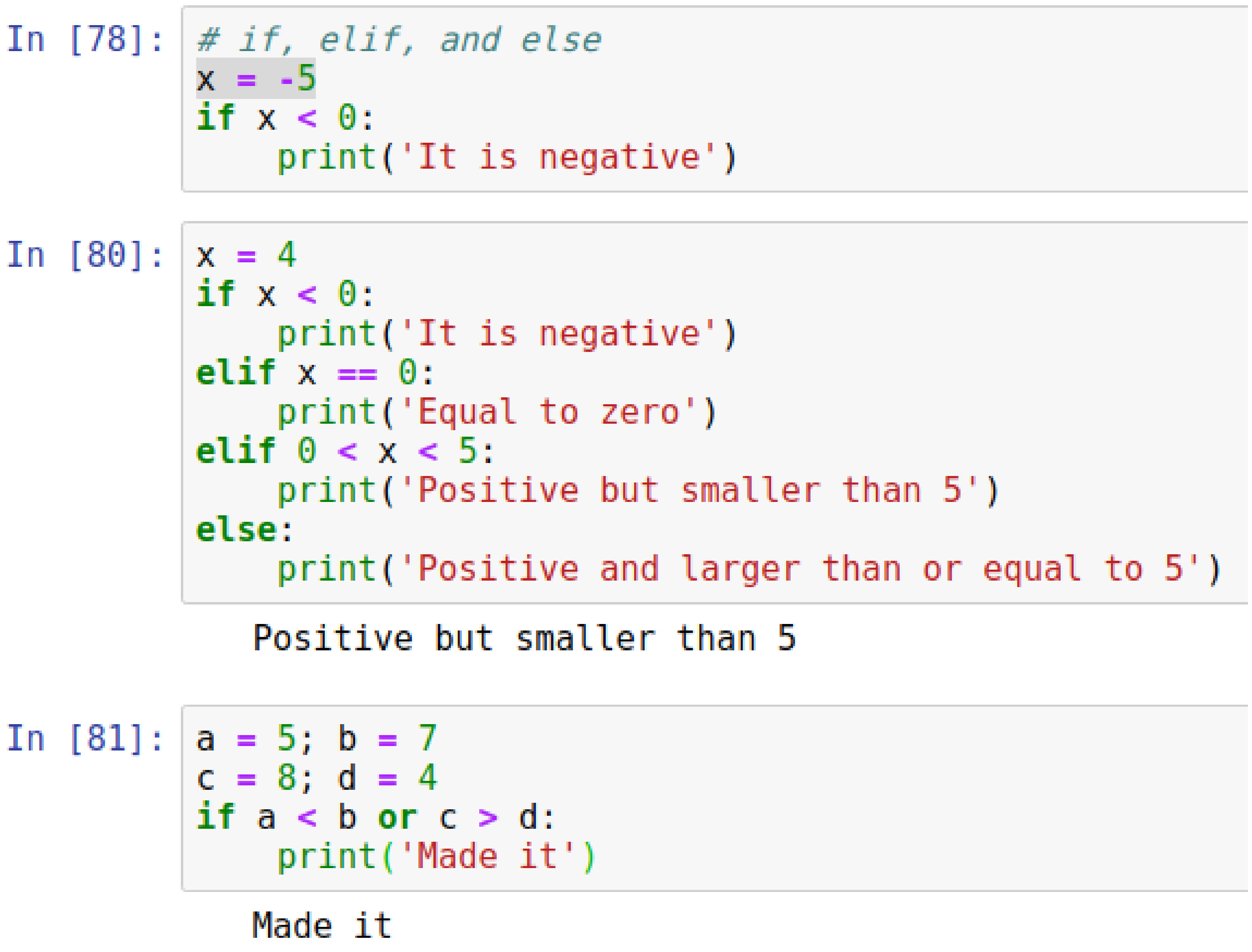
```
In [65]: # The strftime method formats a datetime as a string
         dt.strftime('%m/%d/%Y %H:%M')

Out[65]: '10/29/2011 20:30'

In [66]: # Strings can be converted (parsed) into datetime objects with the strptime function
         datetime.strptime('20091031', '%Y%m%d')

Out[66]: datetime.datetime(2009, 10, 31, 0, 0)

In [68]: dt.replace(minute=0, second=0)

         # Since datetime.datetime is an immutable type, methods like these always produce
         # new objects.

Out[68]: datetime.datetime(2011, 10, 29, 20, 0)

In [72]: # The difference of two datetime objects produces a datetime.timedelta type
         dt2 = datetime(2011, 11, 15, 22, 30)
         delta = dt2 - dt
         delta
         # The output timedelta(17, 7179) indicates that the timedelta encodes an offset of 17
         # days and 7,179 seconds

Out[72]: datetime.timedelta(17, 7179)

In [71]: type(delta)

Out[71]: datetime.timedelta
```

Python Basics : Datetime format specification (ISO C89 compatible)



Type	Description
%Y	Four-digit year
%y	Two-digit year
%m	Two-digit month [01, 12]
%d	Two-digit day [01, 31]
%H	Hour (24-hour clock) [00, 23]
%I	Hour (12-hour clock) [01, 12]
%M	Two-digit minute [00, 59]
%S	Second [00, 61] (seconds 60, 61 account for leap seconds)
%w	Weekday as integer [0 (Sunday), 6]
%U	Week number of the year [00, 53]; Sunday is considered the first day of the week, and days before the first Sunday of the year are "week 0"
%W	Week number of the year [00, 53]; Monday is considered the first day of the week, and days before the first Monday of the year are "week 0"
%z	UTC time zone offset as +HHMM or -HHMM; empty if time zone naive
%F	Shortcut for %Y-%m-%d (e.g., 2012-4-18)
%D	Shortcut for %m/%d/%y (e.g., 04/18/12)

Python Basics : Control Flow

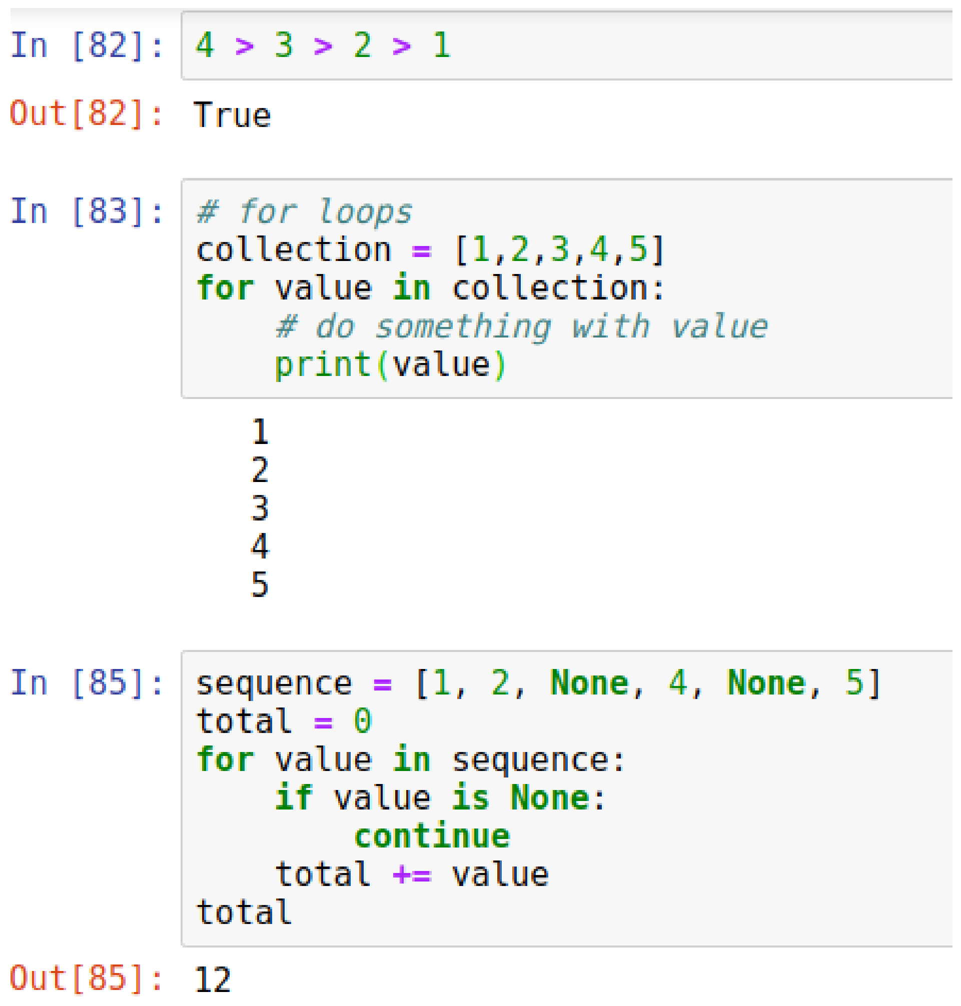


```
In [78]: # if, elif, and else
         x = -5
         if x < 0:
             print('It is negative')
```

```
In [80]: x = 4
         if x < 0:
             print('It is negative')
         elif x == 0:
             print('Equal to zero')
         elif 0 < x < 5:
             print('Positive but smaller than 5')
         else:
             print('Positive and larger than or equal to 5')

         Positive but smaller than 5
```

```
In [81]: a = 5; b = 7
         c = 8; d = 4
         if a < b or c > d:
             print('Made it')

         Made it
```

Python Basics : Control Flow

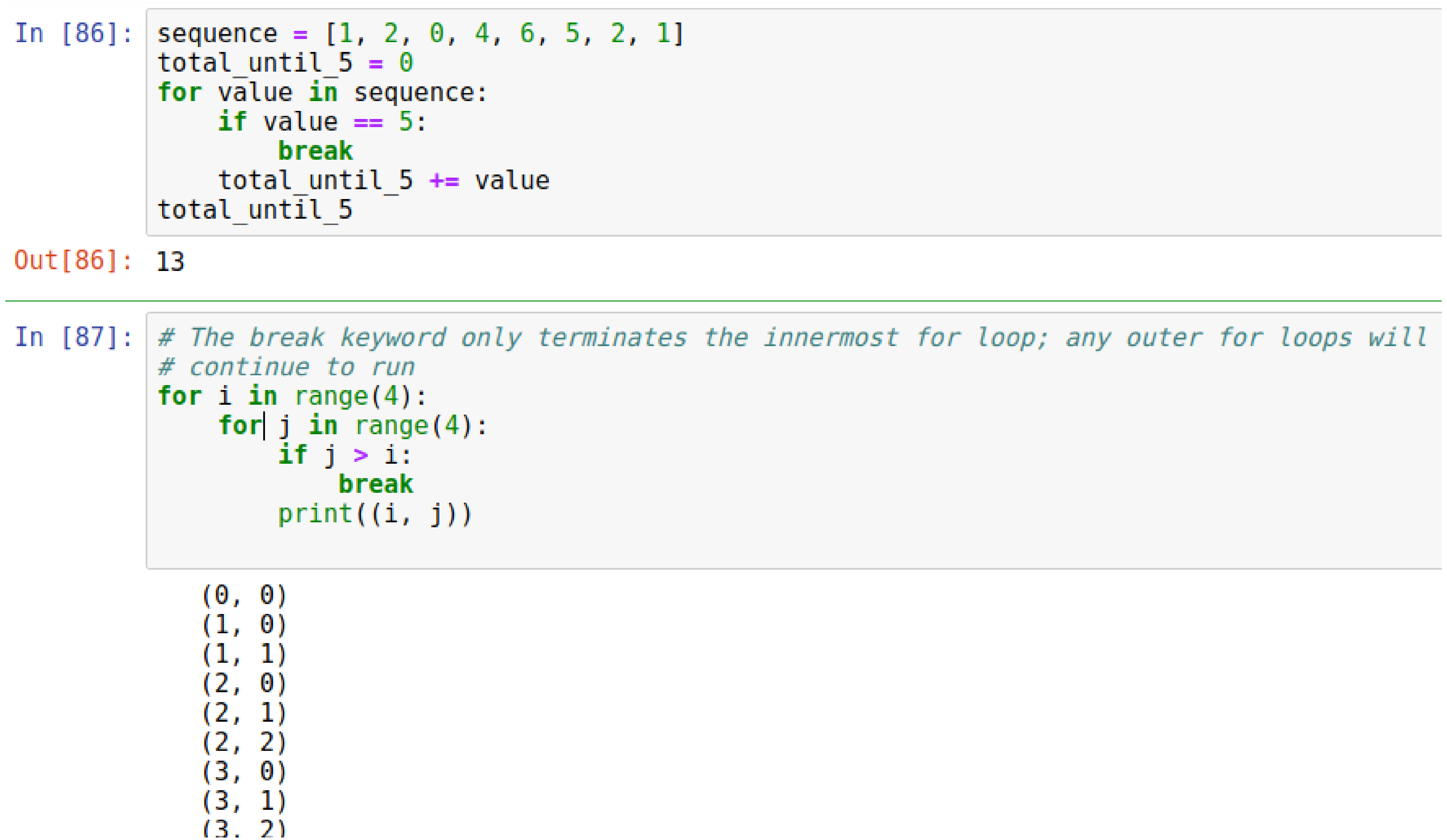


```
In [82]: 4 > 3 > 2 > 1

Out[82]: True
```

```
In [83]: # for loops
         collection = [1,2,3,4,5]
         for value in collection:
             # do something with value
             print(value)

           1
           2
           3
           4
           5
```

```
In [85]: sequence = [1, 2, None, 4, None, 5]
         total = 0
         for value in sequence:
             if value is None:
                 continue
             total += value
         total

Out[85]: 12
```

Python Basics : Control Flow



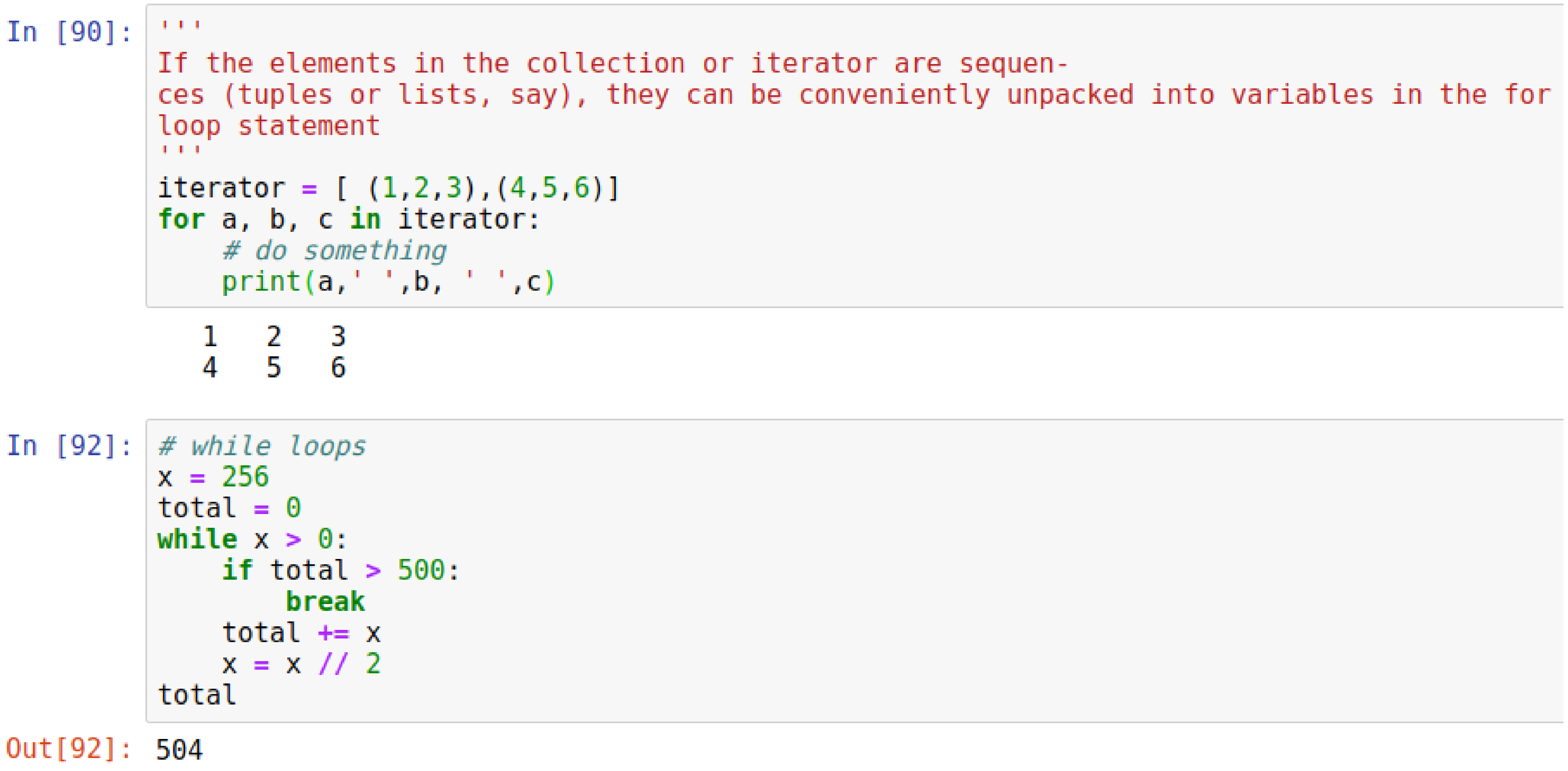
```
In [86]: sequence = [1, 2, 0, 4, 6, 5, 2, 1]
         total_until_5 = 0
         for value in sequence:
             if value == 5:
                 break
             total_until_5 += value
         total_until_5
```

```
Out[86]: 13
```

```
In [87]: # The break keyword only terminates the innermost for loop; any outer for loops will
         # continue to run
         for i in range(4):
             for j in range(4):
                 if j > i:
                     break
                 print((i, j))
```

```
(0, 0)
(1, 0)
(1, 1)
(2, 0)
(2, 1)
(2, 2)
(3, 0)
(3, 1)
(3, 2)
```

Python Basics : Control Flow



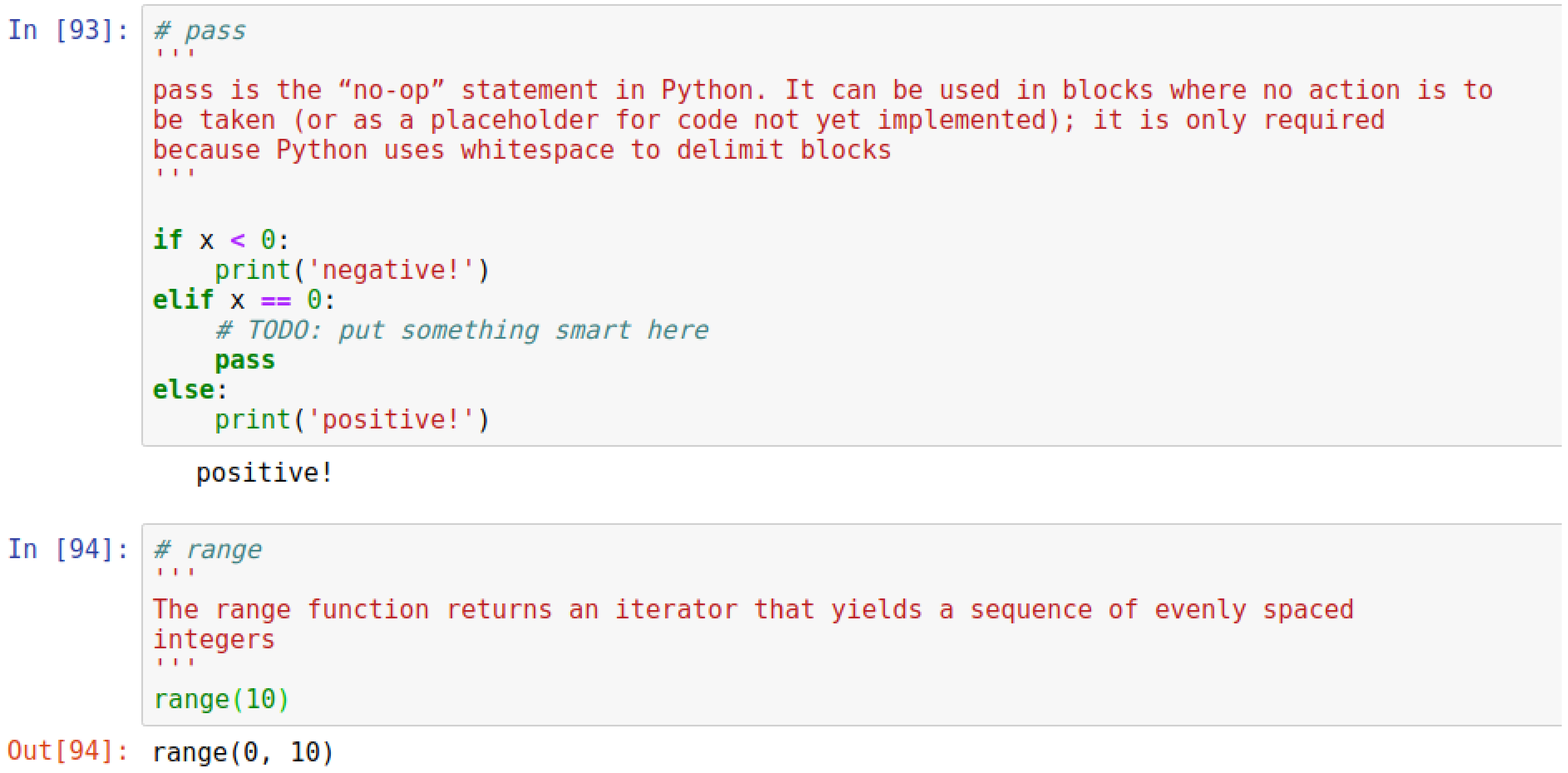
In [90]:
```
'''
If the elements in the collection or iterator are sequen-
ces (tuples or lists, say), they can be conveniently unpacked into variables in the for
loop statement
'''
iterator = [ (1,2,3),(4,5,6)]
for a, b, c in iterator:
    # do something
    print(a,' ',b, ' ',c)
```

```
1   2   3
4   5   6
```

In [92]:
```
# while loops
x = 256
total = 0
while x > 0:
    if total > 500:
        break
    total += x
    x = x // 2
total
```

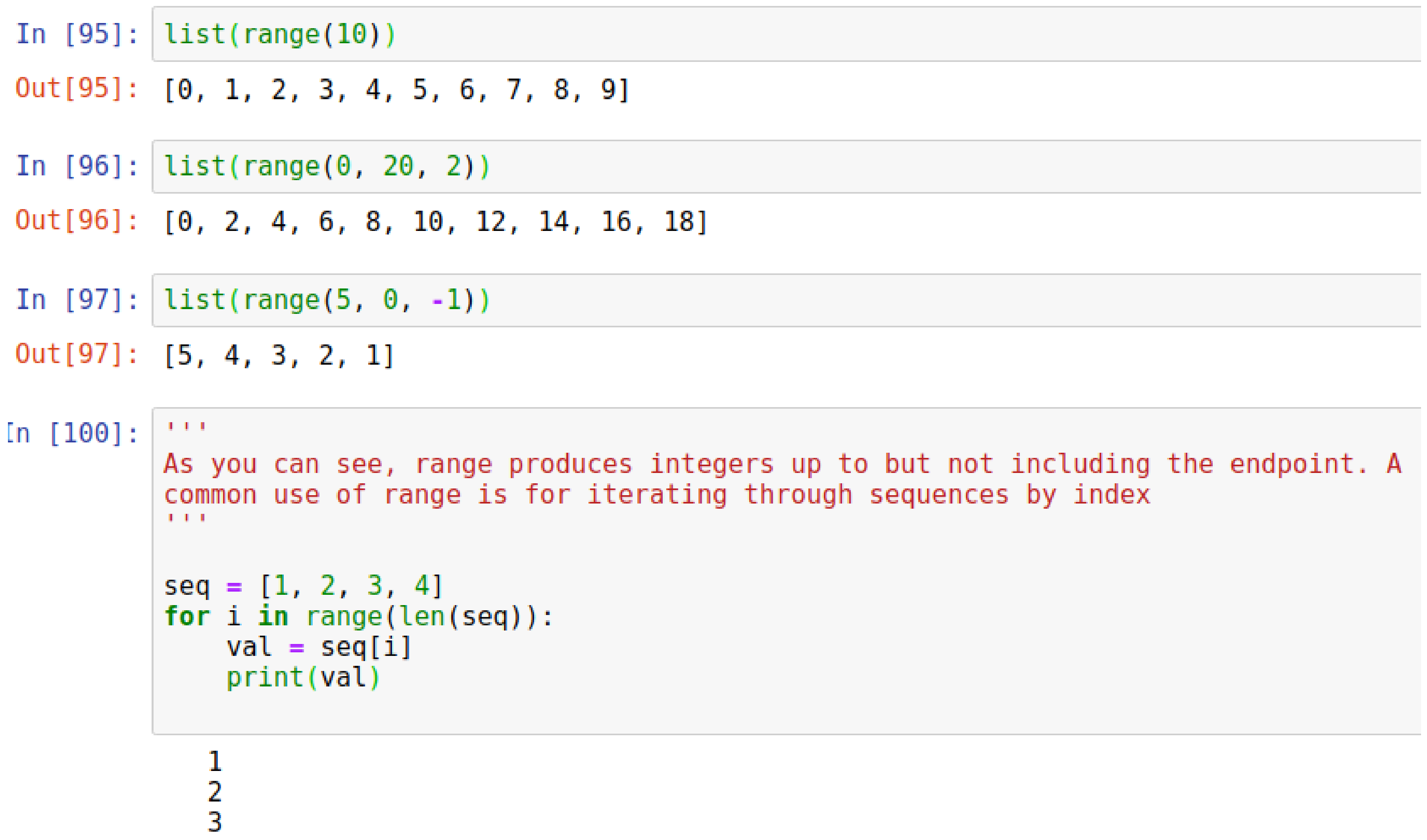
Out[92]:
```
504
```

Python Basics : Control Flow



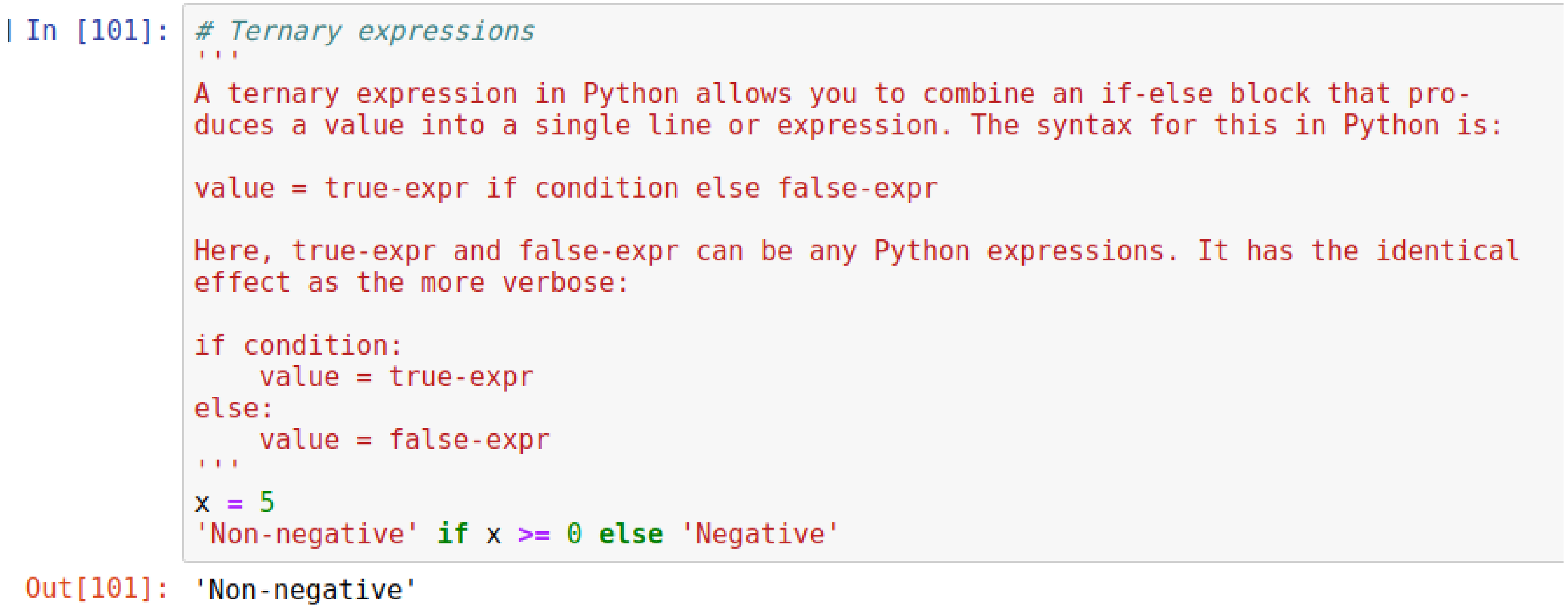
```
In [93]: # pass
         '''
         pass is the “no-op” statement in Python. It can be used in blocks where no action is to
         be taken (or as a placeholder for code not yet implemented); it is only required
         because Python uses whitespace to delimit blocks
         '''

         if x < 0:
             print('negative!')
         elif x == 0:
             # TODO: put something smart here
             pass
         else:
             print('positive!')

           positive!

In [94]: # range
         '''
         The range function returns an iterator that yields a sequence of evenly spaced
         integers
         '''
         range(10)

Out[94]: range(0, 10)
```

Python Basics : Control Flow



```
In [95]: list(range(10))

Out[95]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [96]: list(range(0, 20, 2))

Out[96]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

In [97]: list(range(5, 0, -1))

Out[97]: [5, 4, 3, 2, 1]

In [100]: '''
          As you can see, range produces integers up to but not including the endpoint. A
          common use of range is for iterating through sequences by index
          '''

          seq = [1, 2, 3, 4]
          for i in range(len(seq)):
              val = seq[i]
              print(val)

            1
            2
            3
```

Python Basics : Control Flow



```
In [101]: # Ternary expressions
          '''
          A ternary expression in Python allows you to combine an if-else block that pro-
          duces a value into a single line or expression. The syntax for this in Python is:

          value = true-expr if condition else false-expr

          Here, true-expr and false-expr can be any Python expressions. It has the identical
          effect as the more verbose:

          if condition:
              value = true-expr
          else:
              value = false-expr
          '''
          x = 5
          'Non-negative' if x >= 0 else 'Negative'

Out[101]: 'Non-negative'
```