PYTHON LANGUAGE BASICS

The Python Interpreter

> Python is an *interpreted* language.

> The Python interpreter runs a program by executing one statement at a time.

> The standard interactive Python interpreter can be invoked on the command line with the python command

```
python
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 42
>>> print(x)
42
>>>
```

The Python Interpreter

> Running Python programs is as simple as calling python with a *.py* file as its first argument.

> Suppose we had created *hello_mars.py* with these contents:

```
print('Hello mars')
```

> **python hello_mars.py**

```
Hello mars
```


Scientific Computing

> Tools for data analysis and scientific computing

– IPython

Enhanced Python interpreter

– Jupyter Notebook

web-based code notebooks originally created

within the IPython project

IPython Basics

> You can launch the IPython shell on the command line just like launching the regular Python interpreter:

```
ipython
```

```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)]

Type 'copyright', 'credits' or 'license' for more information

IPython 7.5.0 -- An enhanced Interactive Python. Type '?' for help.


In [1]:
```


ipython

```
In [5]: import numpy as np

In [6]: data = {i : np.random.randn() for i in range(7)}

In [7]: data
Out[7]:
{0: -0.20470765948471295,
 1: 0.47894333805754824,
 2: -0.5194387150567381,
 3: -0.55573030434749,
 4: 1.9657805725027142,
 5: 1.3934058329729904,
 6: 0.09290787674371767}
```

Running Jupyter

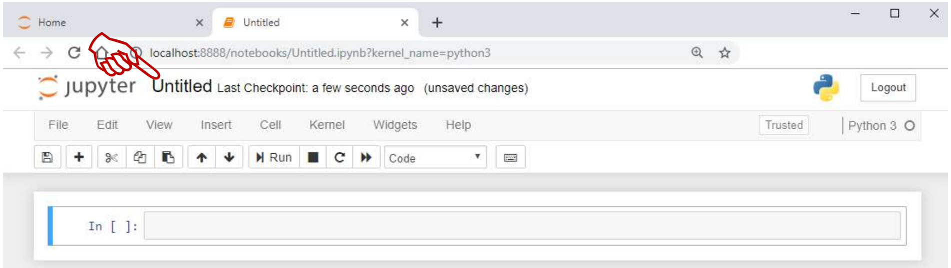
`jupyter notebook`

```
[I 16:54:10.760 NotebookApp] JupyterLab beta preview extension loaded from C:\stage\opt\anaconda3\lib\site-packages\jupyterlab
[I 16:54:10.760 NotebookApp] JupyterLab application directory is C:\stage\opt\anaconda3\share\jupyter\lab
[I 16:54:11.005 NotebookApp] Serving notebooks from local directory: C:\Users\Binnur
[I 16:54:11.005 NotebookApp] 0 active kernels
[I 16:54:11.006 NotebookApp] The Jupyter Notebook is running at:
[I 16:54:11.006 NotebookApp] http://localhost:8888/?token=f449626a667e1288c7552794d8e1df0e1b7d0196c470bec4
[I 16:54:11.006 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 16:54:11.007 NotebookApp]

    Copy/paste this URL into your browser when you connect for the first time,
    to login with a token:
http://localhost:8888/?token=f449626a667e1288c7552794d8e1df0e1b7d0196c470bec4&token=f449626a667e1288c7552794d8e1df0e1b7d0196c470
bec4
[W 16:54:11.018 NotebookApp] 404 GET /api/kernels/11e1e0ee-96c0-4fba-802c-
888db3486004/channels?session_id=7000514bd789457aba8d8566b90c1598 (::1): Kernel does not exist: 11e1e0ee-96c0-4fba-802c-
888db3486004
[W 16:54:11.048 NotebookApp] 404 GET /api/kernels/11e1e0ee-96c0-4fba-802c-
888db3486004/channels?session_id=7000514bd789457aba8d8566b90c1598 (::1) 38.92ms referer=None
[I 16:54:11.167 NotebookApp] Accepting one-time-token-authenticated connection from ::1
```

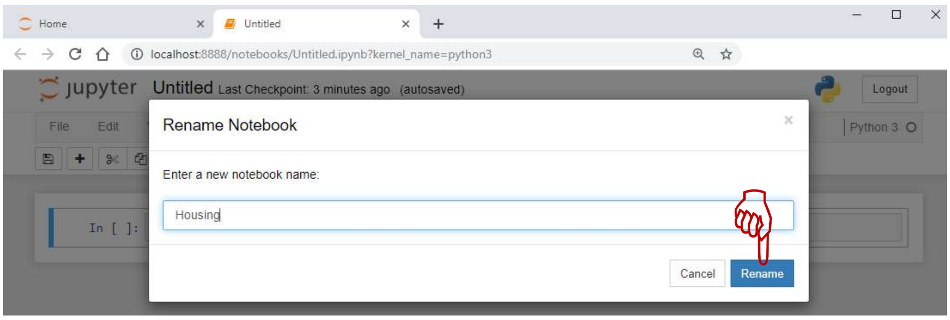

Running Jupyter

Running Jupyter



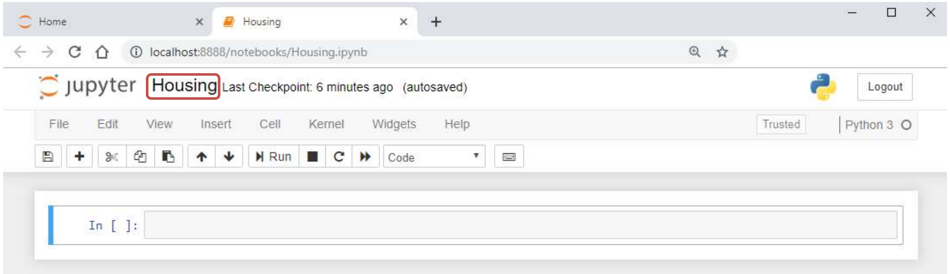
10

Running Jupyter

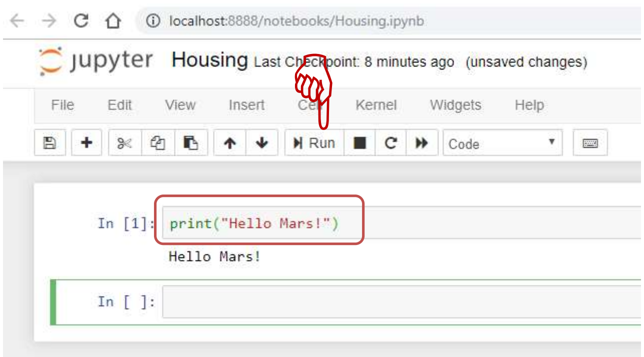


11

Running Jupyter



Running Jupyter



Scalar Types

Type	Description
None	The Python "null" value (only one instance of the None object exists)
str	String type; holds Unicode (UTF-8 encoded) strings
bytes	Raw ASCII bytes (or Unicode encoded as bytes)
float	Double-precision (64-bit) floating-point number (note there is no separate double type)
bool	A True or False value
int	Arbitrary precision signed integer

Numeric types

> The primary Python types for numbers are int and float.

> An int can store arbitrarily large numbers:

```
In [48]: ival = 17239871

In [49]: ival ** 6
Out[49]: 26254519291092456596965462913230729701102721
```

Numeric types

> Floating-point numbers are represented with the Python float type.
  – A double-precision (64-bit) value.

> They can also be expressed with scientific notation:

```
In [50]: fval = 7.243

In [51]: fval2 = 6.78e-5
```

> Integer division not resulting in a whole number will always yield a floating-point number:

```
In [52]: 3 / 2
Out[52]: 1.5
```


Numeric types

> To get C-style integer division, use the floor division operator **//**:

> It drops the fractional part if the result is not a whole number

```
In [53]: 3 // 2
Out[53]: 1
```

Strings

> Python has powerful and flexible built-in string processing capabilities

> You can write *string literals* using either single quotes ' or double quotes ":

```
a = 'one way of writing a string'
b = "another way"
```

> For multiline strings with line breaks, you can use triple quotes, either ''' or """:

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

> The line breaks after """ and after lines are included

```
In [55]: c.count('\n')
Out[55]: 3
```


Strings

> Python strings are immutable; you cannot modify a string:

```
In [56]: a = 'this is a string'

In [57]: a[10] = 'f'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-57-5ca625d1e504> in <module>()
----> 1 a[10] = 'f'
TypeError: 'str' object does not support item assignment

In [58]: b = a.replace('string', 'longer string')

In [59]: b
Out[59]: 'this is a longer string'
```

> After this operation, the variable a is unmodified:

```
In [60]: a
Out[60]: 'this is a string'
```

Strings

> Many Python objects can be converted to a string using the **`str`** function:

```
In [61]: a = 5.6

In [62]: s = str(a)

In [63]: print(s)
5.6
```


Strings

> Strings are a sequence of Unicode characters and can be treated like other sequences, e.g. lists and tuples:

```
In [64]: s = 'python'

In [65]: list(s)
Out[65]: ['p', 'y', 't', 'h', 'o', 'n']

In [66]: s[:3]
Out[66]: 'pyt'
```

slicing

```
In [17]: x = '\u20ba'
         print(x)

         ₺
```

Strings

- The backslash character **** is an *escape character*
- It is used to specify special characters like newline \n or Unicode characters.
- To write a string literal with backslashes, you need to escape them:

```
In [67]: s = '12\\34'

In [68]: print(s)
12\34
```


Strings

- 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*
  - It means that the characters should be interpreted as is:

```
In [69]: s = r'this\has\no\special\characters'

In [70]: s
Out[70]: 'this\\has\\no\\special\\characters'
```

Strings

> Adding two strings together concatenates them and produces a new string:

```
In [71]: a = 'this is the first half '

In [72]: b = 'and this is the second half'

In [73]: a + b
Out[73]: 'this is the first half and this is the second half'
```


Strings

> String templating or formatting is important

> String objects have a format method

– Used to substitute formatted arguments into the string, producing a new string:

```
In [74]: 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.

Strings

> To substitute arguments for these format parameters, pass a sequence of arguments to the **format** method:

```
In [75]: template.format(4.5560, 'Argentine Pesos', 1)
Out[75]: '4.56 Argentine Pesos are worth US$1'
```

> String formatting is a deep topic

– there are multiple methods and numerous options

– tweaks available to control how values are formatted in the resulting string

– To learn more, consult the official Python documentation: https://docs.python.org/3/

Bytes and Unicode

> In modern Python (Python 3.0+), Unicode has become the first-class string type to enable more consistent handling of ASCII and non-ASCII text.

```
In [76]: val = "español"

In [77]: val
Out[77]: 'español'
```

> You can convert this Unicode string to its UTF-8 bytes representation using the **encode** method

```
In [78]: val_utf8 = val.encode('utf-8')

In [79]: val_utf8
Out[79]: b'espa\xc3\xb1ol'

In [80]: type(val_utf8)
Out[80]: bytes
```

Bytes and Unicode

> Assuming you know the Unicode encoding of a **bytes** object, you can go back using the **decode** method:

```
In [81]: val_utf8.decode('utf-8')
Out[81]: 'español'
```

> While it's become preferred to use UTF-8 for any encoding, for historical reasons you may encounter data in any number of different encodings:

```
In [82]: val.encode('latin1')
Out[82]: b'espa\xf1ol'

In [83]: val.encode('utf-16')
Out[83]: b'\xff\xfee\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'

In [84]: val.encode('utf-16le')
Out[84]: b'e\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'
```


Bytes and Unicode

> It is most common to encounter bytes objects in the context of working with files, where implicitly decoding all data to Unicode strings may not be desired.

> Though you may seldom need to do so, you can define your own byte literals by prefixing a string with b:

```
In [85]: bytes_val = b'this is bytes'

In [86]: bytes_val
Out[86]: b'this is bytes'

In [87]: decoded = bytes_val.decode('utf8')

In [88]: decoded  # this is str (Unicode) now
Out[88]: 'this is bytes'
```

Bytes and Unicode

```
x=b"\xf0\x9D\x84\x9e"
len(x)
y = x.decode("utf-8")
len(y)
```

```
1
```

```
print(x,y)
```

```
b'\xf0\x9d\x84\x9e' 𝄞
```

```
tl = "\u20ba"
```

```
print(tl)
```

```
₺
```


Booleans

> The two boolean values in Python are written as **True** and **False**.

> Comparisons and other conditional expressions evaluate to either **True** or **False**.

> Boolean values are combined with the **and** and or keywords:

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

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

Type casting

> The **`str`**, **`bool`**, **`int`**, and **`float`** types are also functions that can be used to cast values to those types:

```
In [91]: s = '3.14159'

In [92]: fval = float(s)

In [93]: type(fval)
Out[93]: float

In [94]: int(fval)
Out[94]: 3

In [95]: bool(fval)
Out[95]: True

In [96]: bool(0)
Out[96]: False
```


None

> **`None`** is the Python null value type.

> If a function does not explicitly return a value, it implicitly returns **`None`**:

```
In [97]: a = None

In [98]: a is None
Out[98]: True

In [99]: b = 5

In [100]: b is not None
Out[100]: True
```

None

> `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
```

> While a technical point, it's worth bearing in mind that `None` is not only a reserved keyword but also a unique instance of `NoneType`:

```
In [101]: type(None)
Out[101]: NoneType
```


Dates and times

> The built-in Python `datetime` module provides `datetime`, `date`, and `time` types.

> The `datetime` type combines the information stored in `date` and `time` and is the most commonly used:

```
In [102]: from datetime import datetime, date, time

In [103]: dt = datetime(2011, 10, 29, 20, 30, 21)

In [104]: dt.day
Out[104]: 29

In [105]: dt.minute
Out[105]: 30
```

Dates and times

> Given a **`datetime`** instance, you can extract the equivalent **`date`** and **`time`** objects by calling methods on the **`datetime`** of the same name:

```
In [106]: dt.date()
Out[106]: datetime.date(2011, 10, 29)
In [107]: dt.time()
Out[107]: datetime.time(20, 30, 21)
```

> The **`strftime`** method formats a datetime as a string:

```
In [108]: dt.strftime('%m/%d/%Y %H:%M')
Out[108]: '10/29/2011 20:30'
```

> Strings can be converted (parsed) into datetime objects with the **`strptime`** function:

```
In [109]: datetime.strptime('20091031', '%Y%m%d')
Out[109]: datetime.datetime(2009, 10, 31, 0, 0)
```


Datetime format specification

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)

Dates and times

> When you are aggregating or otherwise grouping time series data, it will occasionally be useful to replace time fields of a series of **datetimes**

```
In [110]: dt.replace(minute=0, second=0)
Out[110]: datetime.datetime(2011, 10, 29, 20, 0)
```

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

Dates and times

> The difference of two **datetime** objects produces a **datetime.timedelta** type:

```
In [111]: dt2 = datetime(2011, 11, 15, 22, 30)

In [112]: delta = dt2 - dt

In [113]: delta
Out[113]: datetime.timedelta(17, 7179)

In [114]: type(delta)
Out[114]: datetime.timedelta
```

Control Flow

> Python has several built-in keywords for conditional logic, loops, and other standard *control flow* concepts found in other programming languages.
  – if, elif, and else
  – for loops
  – while loops
  – pass
  – range
  – ternary expressions

`if`, `elif`, and `else`

> The **`if`** statement is one of the most well-known control flow statement types.

> It checks a condition that, if **`True`**, evaluates the code in the block that follows:

```
if x < 0:
    print('It's negative')
```

`if`, **`elif`**, and **`else`**

> An **`if`** statement can be optionally followed by one or more **`elif`** blocks and a catch-all **`else`** block if all of the conditions are **`False`**:

```
if x < 0:
    print('It's 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')
```


`if`, **`elif`**, and **`else`**

> If any of the conditions is **`True`**, no further **`elif`** or **`else`** blocks will be reached.

> With a compound condition using **`and`** or **`or`**, conditions are evaluated left to right and will short-circuit:

```
In [117]: a = 5; b = 7

In [118]: c = 8; d = 4

In [119]: if a < b or c > d:
   .....:     print('Made it')
Made it
```

`for` loops

- `for` loops are for iterating over a collection (like a list or tuple) or an iterator.
- The standard syntax for a `for` loop is:

```
for value in collection:
    # do something with value
```


`for` loops

- You can advance a `for` loop to the next iteration, skipping the remainder of the block, using the continue keyword.
- Sum up integers in a list and skips `None` values:

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

for loops

> A **for** loop can be exited altogether with the **break** keyword.

> Sum up elements of the list until a 5 is reached:

```
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
```


for loops

> **break** keyword only terminates the innermost **for** loop

> Any outer for loops will continue to run:

```
In [121]: 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)
(3, 3)
```

`while` loops

> A **`while`** loop specifies a condition and a block of code that is to be executed

– until the condition evaluates to **`False`**

or

– the loop is explicitly ended with **`break`**

```
x = 256
total = 0
while x > 0:
    if total > 500:
        break
    total += x
    x = x // 2
```


`pass`

> **`pass`** is the "no-op" statement in Python.

> It can be used in blocks where

– no action is to be taken

– 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!')
```

range

> The **range** function returns an iterator that yields a sequence of evenly spaced integers:

```
In [122]: range(10)
Out[122]: range(0, 10)

In [123]: list(range(10))
Out[123]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

> Both a start, end, and step (which may be negative) can be given:

```
In [124]: list(range(0, 20, 2))
Out[124]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

In [125]: list(range(5, 0, -1))
Out[125]: [5, 4, 3, 2, 1]
```


range

> **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]
```

`range`

- While you can use functions like `list` to store all the integers generated by `range` in some other data structure, often the default iterator form will be what you want.
- This snippet sums all numbers from 0 to 99,999 that are multiples of 3 or 5:

```
sum = 0
for i in range(100000):
    # % is the modulo operator
    if i % 3 == 0 or i % 5 == 0:
        sum += i
```

- While the range generated can be arbitrarily large, the memory use at any given time may be very small.

Ternary expressions

- A *ternary expression* in Python allows you to combine an if-else block that produces a value into a single line or expression.
- The syntax for this in Python is:

  value = *true-expr* ***if*** *condition* ***else*** *false-expr*

  – *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*

Ternary expressions

> Example

```
In [126]: x = 5

In [127]: 'Non-negative' if x >= 0 else 'Negative'
Out[127]: 'Non-negative'
```


BUILT-IN DATA STRUCTURES, FUNCTIONS, AND FILES

DATA STRUCTURES

Data Structures and Sequences

> Python’s data structures are simple but powerful
  - Tuple
  - List
  - Dictionary
  - Set

> Mastering their use is a critical part of becoming a proficient Python programmer.

Tuple

> A tuple is a ***fixed-length***, ***immutable*** sequence of Python objects.

> The easiest way to create one is with a comma-separated sequence of values:

```
In [1]: tup = 4, 5, 6

In [2]: tup
Out[2]: (4, 5, 6)
```


Tuple

> When you're defining tuples in more complicated expressions, it's often necessary to enclose the values in parentheses

```
In [3]: nested_tup = (4, 5, 6), (7, 8)

In [4]: nested_tup
Out[4]: ((4, 5, 6), (7, 8))
```

Tuple

> You can convert any sequence or iterator to a tuple by invoking tuple:

```
In [5]: tuple([4, 0, 2])
Out[5]: (4, 0, 2)

In [6]: tup = tuple('string')

In [7]: tup
Out[7]: ('s', 't', 'r', 'i', 'n', 'g')
```


Tuple

> Elements can be accessed with square brackets [] as with most other sequence types.

> As in C, C++, Java, and many other languages, sequences are 0-indexed in Python:

```
In [6]: tup = tuple('string')

In [7]: tup
Out[7]: ('s', 't', 'r', 'i', 'n', 'g')

In [8]: tup[0]
Out[8]: 's'
```

Tuple

> Once the tuple is created it's not possible to modify which object is stored in each slot:

```
In [9]: tup = tuple(['foo', [1, 2], True])

In [10]: tup[2] = False
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-c7308343b841> in <module>()
----> 1 tup[2] = False
TypeError: 'tuple' object does not support item assignment
```

> If an object inside a tuple is mutable, such as a list, you can modify it in-place:

```
In [11]: tup[1].append(3)

In [12]: tup
Out[12]: ('foo', [1, 2, 3], True)
```


Tuple

> You can concatenate tuples using the + operator to produce longer tuples:

```
In [13]: (4, None, 'foo') + (6, 0) + ('bar',)
Out[13]: (4, None, 'foo', 6, 0, 'bar')
```

> Multiplying a tuple by an integer, as with lists, has the effect of concatenating together that many copies of the tuple:

```
In [14]: ('foo', 'bar') * 4
Out[14]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

> Note that the objects themselves are not copied, only the references to them.

Unpacking tuples

> If you try to *assign* to a tuple-like expression of variables, Python will attempt to *unpack* the value on the right-hand side of the equals sign:

```
In [15]: tup = (4, 5, 6)

In [16]: a, b, c = tup

In [17]: b
Out[17]: 5
```

> Even sequences with nested tuples can be unpacked:

```
In [18]: tup = 4, 5, (6, 7)

In [19]: a, b, (c, d) = tup

In [20]: d
Out[20]: 7
```


Unpacking tuples

> Swapping in Python

```
In [21]: a, b = 1, 2

In [22]: a
Out[22]: 1

In [23]: b
Out[23]: 2

In [24]: b, a = a, b

In [25]: a
Out[25]: 2

In [26]: b
Out[26]: 1
```

Unpacking tuples

> A common use of variable unpacking is iterating over sequences of tuples or lists:

```
In [27]: seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

In [28]: for a, b, c in seq:
   ....:     print('a={0}, b={1}, c={2}'.format(a, b, c))
a=1, b=2, c=3
a=4, b=5, c=6
a=7, b=8, c=9
```


Unpacking tuples

> The Python language recently acquired some more advanced tuple unpacking to help with situations where you may want to “pluck” a few elements from the beginning of a tuple.

> This uses the special syntax **`*rest`**, which is also used in function signatures to capture an arbitrarily long list of positional arguments:

```
In [29]: values = 1, 2, 3, 4, 5

In [30]: a, b, *rest = values

In [31]: a, b
Out[31]: (1, 2)

In [32]: rest
Out[32]: [3, 4, 5]
```

Tuple methods

> Since the size and contents of a tuple cannot be modified, it is very light on instance methods.

> A particularly useful one is count, which counts the number of occurrences of a value:

```
In [34]: a = (1, 2, 2, 2, 3, 4, 2)

In [35]: a.count(2)
Out[35]: 4
```


List

> In contrast with tuples, lists are variable-length and their contents can be modified in-place.

> You can define them using square brackets [] or using the list type function:

```
In [36]: a_list = [2, 3, 7, None]

In [37]: tup = ('foo', 'bar', 'baz')

In [38]: b_list = list(tup)

In [39]: b_list
Out[39]: ['foo', 'bar', 'baz']

In [40]: b_list[1] = 'peekaboo'

In [41]: b_list
Out[41]: ['foo', 'peekaboo', 'baz']
```

List

> Lists and tuples are semantically similar (though tuples cannot be modified) and can be used interchangeably in many functions.

> The `list` function is frequently used in data processing as a way to materialize an iterator or generator expression:

```
In [42]: gen = range(10)

In [43]: gen
Out[43]: range(0, 10)

In [44]: list(gen)
Out[44]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```


Adding and removing elements

> Elements can be appended to the end of the list with the `append` method:

```
In [45]: b_list.append('dwarf')

In [46]: b_list
Out[46]: ['foo', 'peekaboo', 'baz', 'dwarf']
```

> Using `insert` you can insert an element at a specific location in the list:

```
In [47]: b_list.insert(1, 'red')

In [48]: b_list
Out[48]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

> The insertion index must be between 0 and the length of the list, inclusive

Adding and removing elements

> **insert** is computationally expensive compared with append
  - references to subsequent elements have to be shifted internally to make room for the new element.
  - If you need to insert elements at both the beginning and end of a sequence, you may wish to explore **collections.deque**, a double-ended queue, for this purpose.

Adding and removing elements

> The inverse operation to **insert** is **pop**, which removes and returns an element at a particular index:

```
In [49]: b_list.pop(2)
Out[49]: 'peekaboo'

In [50]: b_list
Out[50]: ['foo', 'red', 'baz', 'dwarf']
```

Adding and removing elements

> Elements can be removed by value with remove

– Locates the first such value

– Removes it from the last

```
In [51]: b_list.append('foo')

In [52]: b_list
Out[52]: ['foo', 'red', 'baz', 'dwarf', 'foo']

In [53]: b_list.remove('foo')

In [54]: b_list
Out[54]: ['red', 'baz', 'dwarf', 'foo']
```


Adding and removing elements

> You can check if a list contains a value using the `in` keyword:

```
In [55]: 'dwarf' in b_list
Out[55]: True
```

> The keyword `not` can be used to negate in:

```
In [56]: 'dwarf' not in b_list
Out[56]: False
```

> Checking whether a list contains a value is a lot slower than doing so with **dicts** and **sets**

– Python makes a ***linear scan*** across the values of the list

– **dicts** and **sets** check (based on hash tables) *in* ***constant time***.

Concatenating and combining lists

> Similar to tuples, adding two lists together with + concatenates them:

```
In [57]: [4, None, 'foo'] + [7, 8, (2, 3)]
Out[57]: [4, None, 'foo', 7, 8, (2, 3)]
```

> If you have a list already defined, you can append multiple elements to it using the extend method:

```
In [58]: x = [4, None, 'foo']

In [59]: x.extend([7, 8, (2, 3)])

In [60]: x
Out[60]: [4, None, 'foo', 7, 8, (2, 3)]
```


Concatenating and combining lists

Note

> List concatenation by addition is a comparatively expensive operation

– A new list must be created and the objects copied over.

> Use extend to append elements to an existing list

– prefer extend if you are building up a large list

```
everything = []
for chunk in list_of_lists:
    everything.extend(chunk)
```

is faster than

```
everything = []
for chunk in list_of_lists:
    everything = everything + chunk
```

Sorting

> You can sort a list in-place (without creating a new object) by calling its sort function:

```
In [61]: a = [7, 2, 5, 1, 3]

In [62]: a.sort()

In [63]: a
Out[63]: [1, 2, 3, 5, 7]
```


Sorting

> sort has a few options that will occasionally come in handy.

> One ability is to pass a secondary *sort key*

– a function that produces a value to use to sort the objects.

> Example: we could sort a collection of strings by their lengths:

```
In [64]: b = ['saw', 'small', 'He', 'foxes', 'six']

In [65]: b.sort(key=len)

In [66]: b
Out[66]: ['He', 'saw', 'six', 'small', 'foxes']
```

Binary search and maintaining a sorted list

> The built-in **bisect** module implements binary search and insertion into a sorted list.

> **bisect.bisect** finds the location where an element should be inserted to keep it sorted

```
In [67]: import bisect

In [68]: c = [1, 2, 2, 2, 3, 4, 7]

In [69]: bisect.bisect(c, 2)
Out[69]: 4

In [70]: bisect.bisect(c, 5)
Out[70]: 6
```


Binary search and maintaining a sorted list

> **bisect.insort** actually inserts the element into that location:

```
In [71]: bisect.insort(c, 6)

In [72]: c
Out[72]: [1, 2, 2, 2, 3, 4, 6, 7]
```

Binary search and maintaining a sorted list

Note

- The `bisect` module functions do not check whether the list is sorted
  - This is computationally expensive
- Using them with an unsorted list will succeed without error but may lead to incorrect results.

Slicing

- You can select sections of most sequence types by using slice notation
- Basic form: [`start`:`stop`]

```
In [73]: seq = [7, 2, 3, 7, 5, 6, 0, 1]

In [74]: seq[1:5]
Out[74]: [2, 3, 7, 5]
```

- Slices can also be assigned to with a sequence:

```
In [75]: seq[3:4] = [6, 3]

In [76]: seq
Out[76]: [7, 2, 3, 6, 3, 5, 6, 0, 1]
```

- The element at the start index is included, the stop index is *not included*
- The number of elements in the result is stop - start.

Slicing

> Either the `start` or `stop` can be omitted

```
In [77]: seq[:5]
Out[77]: [7, 2, 3, 6, 3]

In [78]: seq[3:]
Out[78]: [6, 3, 5, 6, 0, 1]
```

> Negative indices slice the sequence relative to the end:

```
In [79]: seq[-4:]
Out[79]: [5, 6, 0, 1]

In [80]: seq[-6:-2]
Out[80]: [6, 3, 5, 6]
```


Slicing

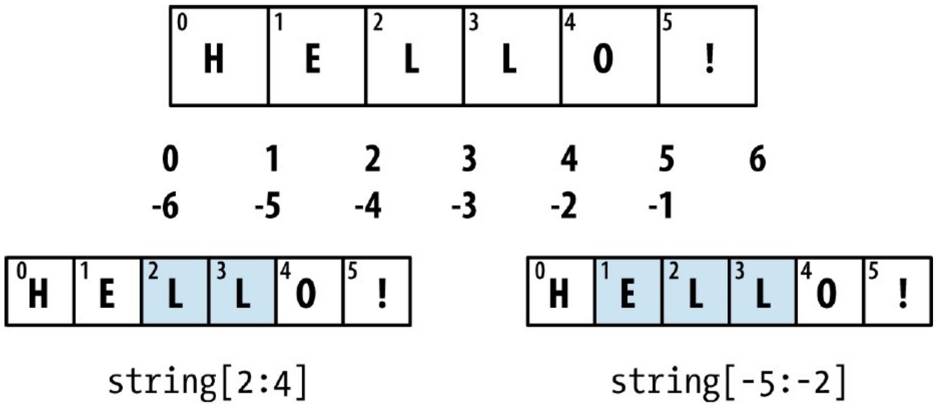
> A `step` can also be used after a second colon to, say, take every other element:

```
In [81]: seq[::2]
Out[81]: [7, 3, 3, 6, 1]
```

> A clever use of this is to pass -1, which has the useful effect of reversing a list or tuple:

```
In [82]: seq[::-1]
Out[82]: [1, 0, 6, 5, 3, 6, 3, 2, 7]
```

Python Slicing Conventions



Built-in Sequence Functions

> Python has a handful of useful sequence functions that you should familiarize yourself with and use at any opportunity:

- enumerate
- sorted
- zip
- reversed

enumerate

> It's common when iterating over a sequence to want to keep track of the index of the current item.

> A do-it-yourself approach would look like:

```
i = 0
for value in collection:
   # do something with value
   i += 1
```

> Since this is so common, Python has a built-in function, enumerate, which returns a sequence of (`i`, `value`) tuples:

```
for i, value in enumerate(collection):
   # do something with value
```


enumerate

> When you are indexing data, a helpful pattern that uses enumerate is computing a dict mapping the values of a sequence (which are assumed to be unique) to their locations in the sequence:

```
In [83]: some_list = ['foo', 'bar', 'baz']

In [84]: mapping = {}

In [85]: for i, v in enumerate(some_list):
   ....:     mapping[v] = i

In [86]: mapping
Out[86]: {'bar': 1, 'baz': 2, 'foo': 0}
```

sorted

> The **`sorted`** function returns a new sorted list from the elements of any sequence:

```
In [87]: sorted([7, 1, 2, 6, 0, 3, 2])
Out[87]: [0, 1, 2, 2, 3, 6, 7]

In [88]: sorted('horse race')
Out[88]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```


zip

> **`zip`** “pairs” up the elements of a number of lists, tuples, or other sequences to create a list of tuples:

```
In [89]: seq1 = ['foo', 'bar', 'baz']

In [90]: seq2 = ['one', 'two', 'three']

In [91]: zipped = zip(seq1, seq2)

In [92]: list(zipped)
Out[92]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

zip

> **zip** can take an arbitrary number of sequences, and the number of elements it produces is determined by the *shortest* sequence:

```
In [93]: seq3 = [False, True]

In [94]: list(zip(seq1, seq2, seq3))
Out[94]: [('foo', 'one', False), ('bar', 'two', True)]
```


zip

> A very common use of **zip** is simultaneously iterating over multiple sequences, possibly also combined with **enumerate**:

```
In [95]: for i, (a, b) in enumerate(zip(seq1, seq2)):
   ....:     print('{0}: {1}, {2}'.format(i, a, b))
   ....:
0: foo, one
1: bar, two
2: baz, three
```

zip

> Given a “zipped” sequence, **`zip`** can be applied in a clever way to “unzip” the sequence.

> Another way to think about this is converting a list of *rows* into a list of *columns*.

> The syntax, which looks a bit magical, is:

```
In [96]: pitchers = [('Nolan', 'Ryan'), ('Roger', 'Clemens'),
   ....:             ('Schilling', 'Curt')]

In [97]: first_names, last_names = zip(*pitchers)

In [98]: first_names
Out[98]: ('Nolan', 'Roger', 'Schilling')

In [99]: last_names
Out[99]: ('Ryan', 'Clemens', 'Curt')
```


reserved

> **`reversed`** iterates over the elements of a sequence in reverse order:

```
In [100]: list(reversed(range(10)))
Out[100]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

> Keep in mind that **`reversed`** is a generator, so it does not create the reversed sequence until materialized

– **`list`**

– **`for`** loop

dict

> dict is likely the most important built-in Python data structure.

> A more common name for it is *hash map* or *associative array*.

> It is a flexibly sized collection of *key-value* pairs, where *key* and *value* are Python objects.

> One approach for creating one is to use curly braces {} and colons to separate keys and values:

```
In [101]: empty_dict = {}

In [102]: d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}

In [103]: d1
Out[103]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```


Functions

> Functions are the primary and most important method of code organization and reuse in Python.

> As a rule of thumb, if you anticipate needing to repeat the same or very similar code more than once, it may be worth writing a reusable function.

> Functions can also help make your code more readable by giving a name to a group of Python statements.

FUNCTIONS

Functions

> Functions are declared with the def keyword and returned from with the return keyword:

```
def my_function(x, y, z=1.5):
    if z > 1:
        return z * (x + y)
    else:
        return z / (x + y)
```

Namespaces, Scope, and Local Functions

> Functions can access variables in two different scopes: *global* and *local*.

> An alternative and more descriptive name describing a variable scope in Python is a *namespace*.

> Any variables that are assigned within a function by default are assigned to the local namespace.

> The local namespace is created when the function is called and immediately populated by the function's arguments.

> After the function is finished, the local namespace is destroyed

FILES AND THE OPERATING SYSTEM

Files and the Operating System

- In practice, you use high-level tools like **pandas.read_csv** to read data files from disk into Python data structures.
- However, it's important to understand the basics of how to work with files in Python.
- Fortunately, it's very simple, which is one reason why Python is so popular for text and file processing

Files and the Operating System

- To open a file for reading or writing, use the built-in open function with either a relative or absolute file path:

```
In [207]: path = 'examples/segismundo.txt'

In [208]: f = open(path)
```

- By default, the file is opened in read-only mode **'r'**.
- You can then treat the file handle **f** like a list and iterate over the lines like so:

```
for line in f:
    pass
```

Files and the Operating System

> The lines come out of the file with the end-of-line (EOL) markers intact, so you'll often see code to get an EOL-free list of lines in a file like:

```
In [209]: lines = [x.rstrip() for x in open(path)]

In [210]: lines
Out[210]:
['Sueña el rico en su riqueza,',
 'que más cuidados le ofrece;',
 '',
 'sueña el pobre que padece',
 'su miseria y su pobreza;',
 '',
 'sueña el que a medrar empieza,',
 'sueña el que afana y pretende,',
 'sueña el que agravia y ofende,',
 '',
 'y en el mundo, en conclusión,',
 'todos sueñan lo que son,',
 'aunque ninguno lo entiende.',
 '']
```


Files and the Operating System

> When you use open to create file objects, it is important to explicitly close the file when you are finished with it.

> Closing the file releases its resources back to the operating system:

```
In [211]: f.close()
```

Files and the Operating System

> One of the ways to make it easier to clean up open files is to use the with statement:

```
In [212]: with open(path) as f:
   .....:     lines = [x.rstrip() for x in f]
```

> This will automatically close the file **`f`** when exiting the with block.

Files and the Operating System

> If we had typed **`f = open(path, 'w')`**, a *new file* at *examples/segismundo.txt* would have been created (be careful!), overwriting any one in its place.

> There is also the **`'x'`** file mode, which creates a writable file but fails if the file path already exists

Files and the Operating System

> For readable files, some of the most commonly used methods are **read**, **seek**, and **tell**.

> **read** returns a certain number of characters from the file.

> What constitutes a "character" is determined by the file's encoding (e.g., UTF-8) or simply raw bytes if the file is opened in binary mode:

```
In [213]: f = open(path)

In [214]: f.read(10)
Out[214]: 'Sueña el r'

In [215]: f2 = open(path, 'rb')  # Binary mode

In [216]: f2.read(10)
Out[216]: b'Sue\xc3\xb1a el '
```


Files and the Operating System

> Even though we read 10 characters from the file, the position is 11 because it took that many bytes to decode 10 characters using the default encoding.

> You can check the default encoding in the **sys** module:

```
In [219]: import sys

In [220]: sys.getdefaultencoding()
Out[220]: 'utf-8'
```

Files and the Operating System

> **seek** changes the file position to the indicated byte in the file:

```
In [221]: f.seek(3)
Out[221]: 3

In [222]: f.read(1)
Out[222]: 'ñ'
```


Python File Modes

Mode	Description
r	Read-only mode
w	Write-only mode; creates a new file (erasing the data for any file with the same name)
x	Write-only mode; creates a new file, but fails if the file path already exists
a	Append to existing file (create the file if it does not already exist)
r+	Read and write
b	Add to mode for binary files (i.e., 'rb' or 'wb')
t	Text mode for files (automatically decoding bytes to Unicode). This is the default if not specified. Add t to other modes to use this (i.e., 'rt' or 'xt')

Important Python File Methods/Attributes

Method	Description
read([size])	Return data from file as a string, with optional size argument indicating the number of bytes to read
readlines([size])	Return list of lines in the file, with optional size argument
write(str)	Write passed string to file
writelines(strings)	Write passed sequence of strings to the file
close()	Close the handle
flush()	Flush the internal I/O buffer to disk
seek(pos)	Move to indicated file position (integer)
tell()	Return current file position as integer
closed	True if the file is closed

Bytes and Unicode with Files

> The default behavior for Python files (whether readable or writable) is ***text mode***

– you intend to work with Python strings (i.e., Unicode).

> This contrasts with ***binary mode***, which you can obtain by appending `b` onto the file mode.

```
In [230]: with open(path) as f:
   .....:     chars = f.read(10)

In [231]: chars
Out[231]: 'Sueña el r'
```

numbers of characters

```
In [232]: with open(path, 'rb') as f:
   .....:     data = f.read(10)

In [233]: data
Out[233]: b'Sue\xc3\xb1a el '
```

exact numbers of bytes

Bytes and Unicode with Files

> Depending on the text encoding, you may be able to decode the bytes to a **`str`** object yourself

– only if each of the encoded Unicode characters is fully formed:

```
In [234]: data.decode('utf8')
Out[234]: 'Sueña el '

In [235]: data[:4].decode('utf8')
---------------------------------------------------------------------------
UnicodeDecodeError                        Traceback (most recent call last)
<ipython-input-235-300e0af10bb7> in <module>()
----> 1 data[:4].decode('utf8')
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc3 in position 3: unexpecte
d end of data
```


Bytes and Unicode with Files

> Text mode, combined with the encoding option of **`open`**, provides a convenient way to convert from one Unicode encoding to another:

```
In [236]: sink_path = 'sink.txt'

In [237]: with open(path) as source:
   .....:     with open(sink_path, 'xt', encoding='iso-8859-1') as sink:
   .....:         sink.write(source.read())

In [238]: with open(sink_path, encoding='iso-8859-1') as f:
   .....:     print(f.read(10))
Sueña el r
```

Bytes and Unicode with Files

- Beware using **seek** when opening files in any mode other than binary.
- If the file position falls in the middle of the bytes defining a Unicode character, then subsequent reads will result in an error:

Bytes and Unicode with Files

```
In [240]: f = open(path)

In [241]: f.read(5)
Out[241]: 'Sueña'

In [242]: f.seek(4)
Out[242]: 4

In [243]: f.read(1)
---------------------------------------------------------------------------
UnicodeDecodeError                        Traceback (most recent call last)
<ipython-input-243-7841103e33f5> in <module>()
----> 1 f.read(1)
/miniconda/envs/book-env/lib/python3.6/codecs.py in decode(self, input, final)
    319         # decode input (taking the buffer into account)
    320         data = self.buffer + input
--> 321         (result, consumed) = self._buffer_decode(data, self.errors, final
)
    322         # keep undecoded input until the next call
    323         self.buffer = data[consumed:]
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb1 in position 0: invalid s
tart byte

In [244]: f.close()
```