Handout Week 1: Python Basics

Dr Ian (Yinglong) He

Click to download the Jupyter Notebook files (.ipynb)

based on the notes by Jake VanderPlas

Contents

1 Introduction

Created by Guido van Rossum and first released in the early 1990s, Python has since become an essential tool for many **programmers, engineers, researchers, and data scientists** across academia and industry. For example, as a researcher focused on building and promoting the free open tools for data-intensive science, I've found Python to be a near-perfect fit for the types of problems I face day to day, whether it's extracting meaning from large datasets, scraping and munging data sources from websites, or automating day-to-day research tasks.

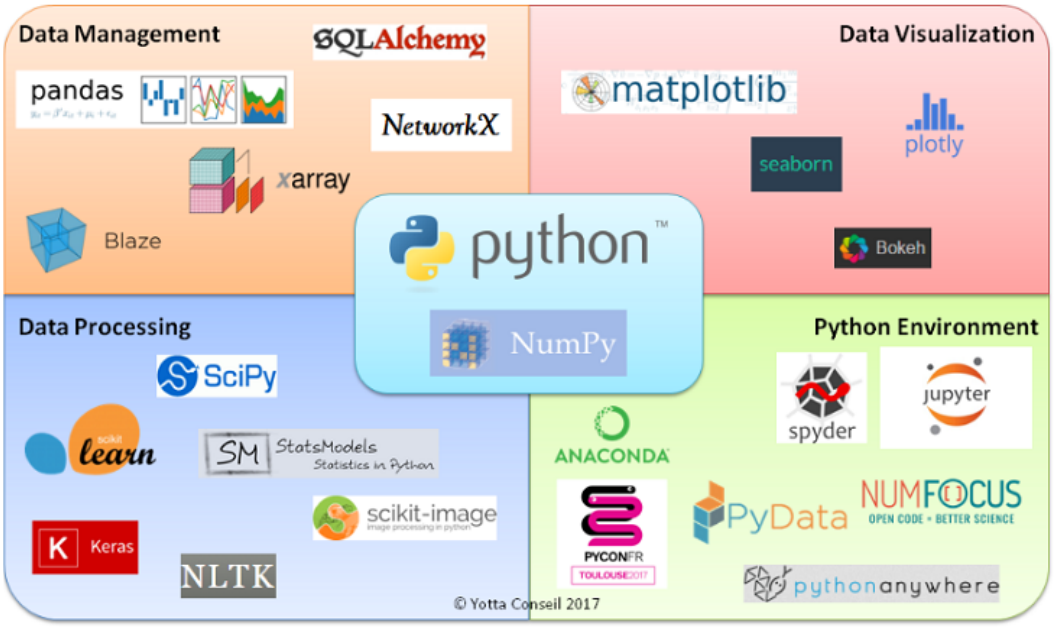
1.1 What is Python?

Python is a computer programming language often used to build websites and software, automate tasks, and conduct data analysis. Python is a general-purpose language, meaning it can be used to create a variety of different programs and isn't specialized for any specific problems. This versatility, along with its beginner-friendliness, has made it one of the most-used programming languages today.

1.2 Python Eocsystem

The appeal of Python is in its simplicity and beauty, as well as the convenience of the large ecosystem of domain-specific tools that have been built on top of it. For example, most of the Python code in scientific computing and data science is built around a group of mature and useful packages:

- NumPy provides efficient storage and computation for multi-dimensional **data arrays**.
- SciPy contains a wide array of **numerical tools** such as numerical integration and interpolation.
- Pandas provides a DataFrame object along with a powerful set of methods to manipulate, filter, group, and transform **heterogeneous and tabular data**.
- Matplotlib provides a useful interface for creation of publication-quality **plots and figures**.
- Scikit-Learn provides a uniform toolkit for applying common **machine learning** algorithms to data.
- IPython/Jupyter provides an enhanced terminal and an **interactive notebook environment** that is useful for exploratory analysis, as well as creation of interactive, executable documents. For example, the manuscript for this report was composed entirely in Jupyter notebooks.



(The Python data science ecosystem. Source: Atrebas)

No less important are the numerous other tools and packages which accompany these: if there is a scientific or data analysis task you want to perform, chances are someone has written a package that will do it for you.

This handout provides a tour of the essential features of the Python language, which are necessary to tap into the power of this data science ecosystem.

1.3 Python vs Matlab

Matlab is a mathematical and technical purpose programming language that includes arrays, linear algebra, and matrices. It is broadly known as a high-quality environment for any task. On the other hand, Python is a general-purpose language and is becoming popular daily due to its ease and functionality.

Python possesses **all the computational power** of Matlab for science and computation to develop new applications easily, but it is different from Matlab. Advantages of Python include:

- It has a large standard **library**.
- It is **free**.
- It can be easily shared or it is very **portable**.
- It has a large **community**.
- It has **open-source** algorithms.
- Its code is more **compact and readable**.

		Matlab	Python
Throw-away	System modelling	Great	Can replace Matlab
	Signal processing	Abundance of toolboxes	
	Data visualization	Excel could be quicker than both	
Infrastructure	Debugging (bypassing)	Matlab is not commonly used in these domains	Great
	Software testing		
	Hardware test automation		
	Data science, machine learning, shiny stuff	N/A	Easy to use

(Comparison of Python and Matlab. Source: Calltutors Team)

Python

```
>>> import numpy as np

# Create row vector
>>> row = np.array([1, 2, 3])
>>> row
array([1, 2, 3])

# Transpose
>>> col = row.T

# Compute inner product
>>> inner = np.dot(row,col)
>>> inner
14

# Compute outer product
>>> outer = np.dot(col,row)
>>> outer
14
```

MATLAB

```
% Create row vector
>> row = [1 2 3]
row =
     1     2     3

% Transpose
>> col = row';

% Compute inner product
>> inner = row*col
inner =
    14

% Compute outer product
>> outer = col*row
outer =
     1     2     3
     2     4     6
     3     6     9
```

(Array operations in Python and Matlab. Source: Asem Elshimi)

1.4 Python 2 vs Python 3

This course uses the syntax of Python 3, which contains language enhancements that are not compatible with the 2.x series of Python. Though Python 3.0 was first released in 2008, adoption has been relatively slow, particularly in the scientific and web development communities. This is primarily because it took some time for many of the essential third-party packages and toolkits to be made compatible with the new language internals. Since early 2014, however, stable releases of the most important tools in the data science ecosystem have been fully compatible

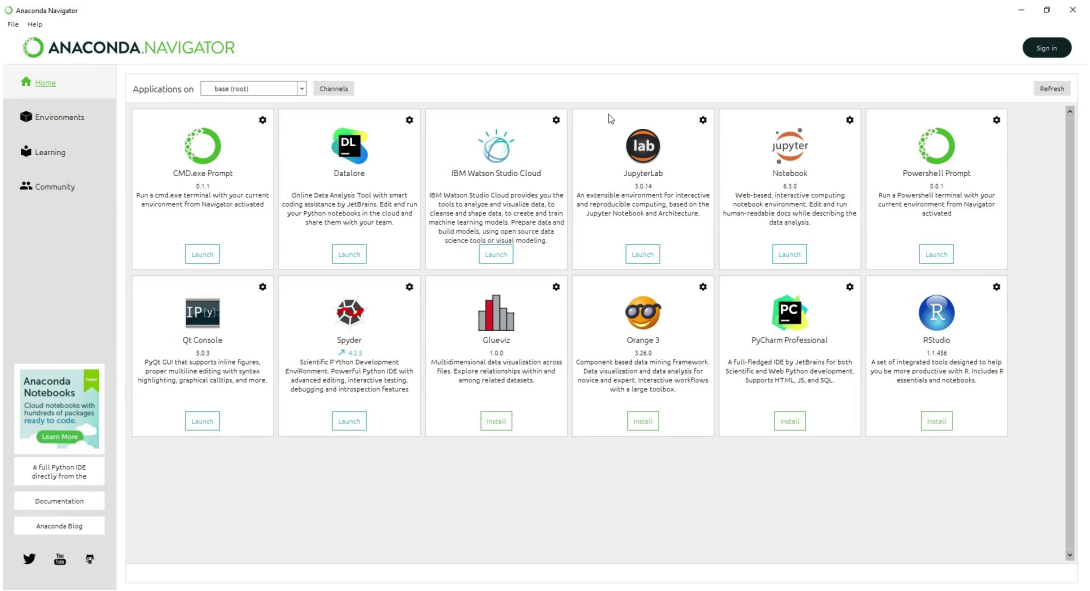
with both Python 2 and 3, and so this book will use the newer Python 3 syntax. However, the vast majority of code snippets in this book will also work without modification in Python 2: in cases where a Py2-incompatible syntax is used, I will make every effort to note it explicitly.

1.5 Installation Considerations

Installing Python and the suite of libraries that enable scientific computing is straightforward. This section will outline some of the considerations when setting up your computer.

Though there are various ways to install Python, the one I would suggest for use in data science is the Anaconda distribution, which works similarly whether you use Windows, Linux, or Mac OS X. The Anaconda distribution comes in two flavors:

- Anaconda includes both Python and conda, and additionally bundles a suite of other pre-installed packages geared toward scientific computing. Because of the size of this bundle, expect the installation to consume several gigabytes of disk space.
- Miniconda gives you the Python interpreter itself, along with a command-line tool called `conda` which operates as a cross-platform package manager geared toward Python packages, similar in spirit to the apt or yum tools that Linux users might be familiar with.



(Anaconda Navigator)

I suggest starting with Anaconda, which is a perfect platform for beginners and has advantages including:

- It lets us create environments to install libraries and packages. This environment is completely independent of the operating system or admin libraries. This means we can create user-level environments with custom versions of libraries for specific projects, which helps us port the project across operating systems with minimal effort.
- It can have multiple environments with different versions of Python and supporting libraries. This way, any version mismatch can be avoided and is not affected by existing packages and libraries of the operating system.
- It comes preloaded with most of necessary packages and libraries for data-science-related tasks

To get started, download and install the Anaconda distribution – make sure to choose a version with Python 3.

2 Basic Python Syntax

Python was originally developed as a teaching language, but its ease of use and clean syntax have led it to be embraced by beginners and experts alike. The cleanliness of Python's syntax has led some to call it "executable pseudocode", and indeed my own experience has been that it is often much easier to read and understand a Python script than to read a similar script written in, say, C. Here we'll begin to discuss the main features of Python's syntax.

Syntax refers to the structure of the language (i.e., what constitutes a correctly-formed program). For the time being, we'll not focus on the semantics – the meaning of the words and symbols within the syntax – but will return to this at a later point.

Consider the following code example:

[1]:
```
# set the midpoint
midpoint = 5

# make two empty lists
lower = []; upper = []

# split the numbers into lower and upper
for i in range(10):
    if (i < midpoint):
        lower.append(i)
    else:
        upper.append(i)

print("lower:", lower)
print("upper:", upper)
```

```
lower: [0, 1, 2, 3, 4]
upper: [5, 6, 7, 8, 9]
```

This script is a bit silly, but it compactly illustrates several of the important aspects of Python syntax. Let's walk through it and discuss some of the syntactical features of Python

2.1 Comments Are Marked by

The script starts with a comment:

```
# set the midpoint
```

Comments in Python are indicated by a pound sign (#), and anything on the line following the pound sign is ignored by the interpreter. This means, for example, that you can have stand-alone comments like the one just shown, as well as inline comments that follow a statement. For example:

```
x += 2  # shorthand for x = x + 2
```

Python does not have any syntax for multi-line comments, such as the `/* ... */` syntax used in C and C++, though multi-line strings are often used as a replacement for multi-line comments.

2.2 End-of-Line Terminates a Statement

The next line in the script is

```
midpoint = 5
```

This is an assignment operation, where we've created a variable named `midpoint` and assigned it the value `5`. Notice that the end of this statement is simply marked by the end of the line. This is in contrast to languages like C and C++, where every statement must end with a semicolon (`;`).

In Python, if you'd like a statement to continue to the next line, it is possible to use the "\" marker to indicate this:

[2]:
```
x = 1 + 2 + 3 + 4 +\
    5 + 6 + 7 + 8
```

It is also possible to continue expressions on the next line within parentheses, without using the "\" marker:

[3]:
```
x = (1 + 2 + 3 + 4 +
     5 + 6 + 7 + 8)
```

Most Python style guides recommend the second version of line continuation (within parentheses) to the first (use of the "\" marker).

2.3 Semicolon Can Optionally Terminate a Statement

Sometimes it can be useful to put multiple statements on a single line. The next portion of the script is

```
lower = []; upper = []
```

This shows the example of how the semicolon (`;`) familiar in C can be used optionally in Python to put two statements on a single line. Functionally, this is entirely equivalent to writing

```
lower = []
upper = []
```

Using a semicolon to put multiple statements on a single line is generally discouraged by most Python style guides, though occasionally it proves convenient.

2.4 Indentation: Whitespace Matters!

Next, we get to the main block of code:

```
for i in range(10):
    if i < midpoint:
        lower.append(i)
    else:
        upper.append(i)
```

This is a compound control-flow statement including a loop and a conditional – we'll look at these types of statements in a moment. For now, consider that this demonstrates what is perhaps the most controversial feature of Python's syntax: whitespace is meaningful!

In programming languages, a *block* of code is a set of statements that should be treated as a unit. In C, for example, code blocks are denoted by curly braces:

```c
// C code
for(int i=0; i<100; i++)
   {
      // curly braces indicate code block
      total += i;
   }
```

In Python, code blocks are denoted by *indentation*:

```python
for i in range(100):
    # indentation indicates code block
    total += i
```

In Python, indented code blocks are always preceded by a colon (`:`) on the previous line.

The use of indentation helps to enforce the uniform, readable style that many find appealing in Python code. But it might be confusing to the uninitiated; for example, the following two snippets will produce different results:

```
>>> if x < 4:          >>> if x < 4:
...     y = x * 2      ...     y = x * 2
...     print(x)       ... print(x)
```

In the snippet on the left, `print(x)` is in the indented block, and will be executed only if `x` is less than `4`. In the snippet on the right `print(x)` is outside the block, and will be executed regardless of the value of `x`!

Python's use of meaningful whitespace often is surprising to programmers who are accustomed to other languages, but in practice it can lead to much more consistent and readable code than languages that do not enforce indentation of code blocks. If you find Python's use of whitespace disagreeable, I'd encourage you to give it a try: as I did, you may find that you come to appreciate it.

Finally, you should be aware that the *amount* of whitespace used for indenting code blocks is up to the user, as long as it is consistent throughout the script. By convention, most style guides recommend to indent code blocks by four spaces, and that is the convention we will follow in this report. Note that many text editors like Emacs and Vim contain Python modes that do four-space indentation automatically.

2.5 Whitespace *Within* Lines Does Not Matter

While the mantra of *meaningful whitespace* holds true for whitespace *before* lines (which indicate a code block), white space *within* lines of Python code does not matter. For example, all three of these expressions are equivalent:

[4]:
```python
x=1+2
x = 1 + 2
x             =         1     +                 2
```

Abusing this flexibility can lead to issues with code readibility – in fact, abusing white space is often one of the primary means of intentionally obfuscating code (which some people do for sport). Using whitespace effectively can lead to much more readable code, especially in cases where operators follow each other – compare the following two expressions for exponentiating by a negative number:

```python
x=10**-2
```

to

```
x = 10 ** -2
```

I find the second version with spaces much more easily readable at a single glance. Most Python style guides recommend using a single space around binary operators, and no space around unary operators.

2.6 Parentheses Are for Grouping or Calling

In the previous code snippet, we see two uses of parentheses. First, they can be used in the typical way to group statements or mathematical operations:

[5]:
```
2 * (3 + 4)
```

[5]:
```
14
```

They can also be used to indicate that a *function* is being called. In the next snippet, the `print()` function is used to display the contents of a variable (see the sidebar). The function call is indicated by a pair of opening and closing parentheses, with the *arguments* to the function contained within:

[6]:
```
print('first value:', 1)
```

```
first value: 1
```

[7]:
```
print('second value:', 2)
```

```
second value: 2
```

Some functions can be called with no arguments at all, in which case the opening and closing parentheses still must be used to indicate a function evaluation. An example of this is the `sort` method of lists:

[8]:
```
L = [4,2,3,1]
L.sort()
print(L)
```

```
[1, 2, 3, 4]
```

The "`()`" after `sort` indicates that the function should be executed, and is required even if no arguments are necessary.

2.7 Aside: A Note on the `print()` Function

Above we used the example of the `print()` function. The `print()` function is one piece that has changed between Python *2.x* and Python *3.x*. In Python 2, `print` behaved as a statement: that is, you could write

```
# Python 2 only!
>> print "first value:", 1
first value: 1
```

For various reasons, the language maintainers decided that in Python 3 `print()` should become a function, so we now write

```
# Python 3 only!
>>> print("first value:", 1)
first value: 1
```

This is one of the many backward-incompatible constructs between Python 2 and 3. As of the writing of this book, it is common to find examples written in both versions of Python, and the presence of the `print` statement rather than the `print()` function is often one of the first signs that you're looking at Python 2 code.

3 Scalar Types

When discussing Python variables and objects, we mentioned the fact that all Python objects have type information attached. Here we'll briefly walk through the built-in simple types offered by Python. We say "simple types" to contrast with several compound types, which will be discussed in the following section.

Python's simple types are summarized in the following table:

Python Scalar Types

Type	Example	Description
`int`	`x = 1`	integers (i.e., whole numbers)
`float`	`x = 1.0`	floating-point numbers (i.e., real numbers)
`complex`	`x = 1 + 2j`	Complex numbers (i.e., numbers with real and imaginary part)
`bool`	`x = True`	Boolean: True/False values
`str`	`x = 'abc'`	String: characters or text
`NoneType`	`x = None`	Special object indicating nulls

We'll take a quick look at each of these in turn.

3.1 Integers

The most basic numerical type is the integer. Any number without a decimal point is an integer:

[9]:
```
x = 1
type(x)
```

[9]:
```
int
```

Python integers are actually quite a bit more sophisticated than integers in languages like `C`. C integers are fixed-precision, and usually overflow at some value (often near 2^{31} or 2^{63}, depending on your system). Python integers are variable-precision, so you can do computations that would overflow in other languages:

[10]:
```
2 ** 200
```

[10]:
```
1606938044258990275541962092341162602522202993782792835301376
```

Another convenient feature of Python integers is that by default, division up-casts to floating-point type:

[11]:
```
5 / 2
```

[11]:
```
2.5
```

Note that this upcasting is a feature of Python 3; in Python 2, like in many statically-typed languages such as C, integer division truncates any decimal and always returns an integer:

```
# Python 2 behavior
>>> 5 / 2
2
```

To recover this behavior in Python 3, you can use the floor-division operator:

[12]:
```
5 // 2
```

[12]:
```
2
```

Finally, note that although Python *2.x* had both an `int` and `long` type, Python 3 combines the behavior of these two into a single `int` type.

3.2 Floating-Point Numbers

The floating-point type can store fractional numbers. They can be defined either in standard decimal notation, or in exponential notation:

[13]:
```
x = 0.000005
y = 5e-6
print(x == y)
```

```
True
```

[14]:
```
x = 1400000.00
y = 1.4e6
print(x == y)
```

```
True
```

In the exponential notation, the `e` or `E` can be read "...times ten to the...", so that `1.4e6` is interpreted as 1.4×10^6.

An integer can be explicitly converted to a float with the `float` constructor:

[15]:
```
float(1)
```

[15]:
```
1.0
```

3.2.1 Aside: Floating-point precision

One thing to be aware of with floating point arithmetic is that its precision is limited, which can cause equality tests to be unstable. For example:

[16]:
```
0.1 + 0.2 == 0.3
```

[16]:
```
False
```

Why is this the case? It turns out that it is not a behavior unique to Python, but is due to the fixed-precision format of the binary floating-point storage used by most, if not all, scientific computing platforms. All programming languages using floating-point numbers store them in a

fixed number of bits, and this leads some numbers to be represented only approximately. We can see this by printing the three values to high precision:

[17]:
```
print("0.1 = {0:.17f}".format(0.1))
print("0.2 = {0:.17f}".format(0.2))
print("0.3 = {0:.17f}".format(0.3))
```

```
0.1 = 0.10000000000000001
0.2 = 0.20000000000000001
0.3 = 0.29999999999999999
```

We're accustomed to thinking of numbers in decimal (base-10) notation, so that each fraction must be expressed as a sum of powers of 10:

$$1/8 = 1 \cdot 10^{-1} + 2 \cdot 10^{-2} + 5 \cdot 10^{-3}$$

In the familiar base-10 representation, we represent this in the familiar decimal expression: 0.125.

Computers usually store values in binary notation, so that each number is expressed as a sum of powers of 2:

$$1/8 = 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}$$

In a base-2 representation, we can write this 0.001_2, where the subscript 2 indicates binary notation. The value $0.125 = 0.001_2$ happens to be one number which both binary and decimal notation can represent in a finite number of digits.

In the familiar base-10 representation of numbers, you are probably familiar with numbers that can't be expressed in a finite number of digits. For example, dividing 1 by 3 gives, in standard decimal notation:

$$1/3 = 0.33333333\cdots$$

The 3s go on forever: that is, to truly represent this quotient, the number of required digits is infinite!

Similarly, there are numbers for which binary representations require an infinite number of digits. For example:

$$1/10 = 0.00011001100110011\cdots_2$$

Just as decimal notation requires an infinite number of digits to perfectly represent 1/3, binary notation requires an infinite number of digits to represent 1/10. Python internally truncates these representations at 52 bits beyond the first nonzero bit on most systems.

This rounding error for floating-point values is a necessary evil of working with floating-point numbers. The best way to deal with it is to always keep in mind that floating-point arithmetic is approximate, and *never* rely on exact equality tests with floating-point values.

3.3 Complex Numbers

Complex numbers are numbers with real and imaginary (floating-point) parts. We've seen integers and real numbers before; we can use these to construct a complex number:

[18]:
```
complex(1, 2)
```

[18]:
```
(1+2j)
```

Alternatively, we can use the "`j`" suffix in expressions to indicate the imaginary part:

[19]:
```
1 + 2j
```

[19]:
```
(1+2j)
```

Complex numbers have a variety of interesting attributes and methods, which we'll briefly demonstrate here:

[20]:
```
c = 3 + 4j
```

[21]:
```
c.real  # real part
```

[21]:
```
3.0
```

[22]:
```
c.imag  # imaginary part
```

[22]:
```
4.0
```

[23]:
```
c.conjugate()  # complex conjugate
```

[23]:
```
(3-4j)
```

[24]:
```
abs(c)  # magnitude, i.e. sqrt(c.real ** 2 + c.imag ** 2)
```

[24]:
```
5.0
```

3.4 String Type

Strings in Python are created with single or double quotes:

[25]:
```
message = "what do you like?"
response = 'spam'
```

Python has many extremely useful string functions and methods; here are a few of them:

[26]:
```
# length of string
len(response)
```

[26]:
```
4
```

[27]:
```
# Make upper-case. See also str.lower()
response.upper()
```

[27]:
```
'SPAM'
```

[28]:
```
# Capitalize. See also str.title()
message.capitalize()
```

[28]:
```
'What do you like?'
```

[29]:
```
# concatenation with +
message + response
```

[29]:
```
'what do you like?spam'
```

[30]:
```
# multiplication is multiple concatenation
5 * response
```

[30]:
```
'spamspamspamspamspam'
```

[31]:
```
# Access individual characters (zero-based indexing)
message[0]
```

[31]:
```
'w'
```

3.5 None Type

Python includes a special type, the `NoneType`, which has only a single possible value: `None`. For example:

[32]:
```
type(None)
```

[32]:
```
NoneType
```

You'll see `None` used in many places, but perhaps most commonly it is used as the default return value of a function. For example, the `print()` function in Python 3 does not return anything, but we can still catch its value:

[33]:
```
return_value = print('abc')
```

```
abc
```

[34]:
```
print(return_value)
```

```
None
```

Likewise, any function in Python with no return value is, in reality, returning `None`.

3.6 Boolean Type

The Boolean type is a simple type with two possible values: `True` and `False`, and is returned by comparison operators discussed previously:

[35]:
```
result = (4 < 5)
result
```

[35]:
```
True
```

[36]:
```
type(result)
```

[36]:
```
bool
```

Keep in mind that the Boolean values are case-sensitive: unlike some other languages, `True` and `False` must be capitalized!

[37]:
```
print(True, False)
```

```
True False
```

Booleans can also be constructed using the `bool()` object constructor: values of any other type can be converted to Boolean via predictable rules. For example, any numeric type is False if equal to zero, and True otherwise:

[38]:
```
bool(2014)
```

[38]:
```
True
```

[39]:
```
bool(0)
```

[39]:
```
False
```

[40]:
```
bool(3.1415)
```

[40]:
```
True
```

The Boolean conversion of `None` is always False:

[41]:
```
bool(None)
```

[41]:
```
False
```

For strings, `bool(s)` is False for empty strings and True otherwise:

[42]:
```
bool("")
```

[42]:
```
False
```

[43]:
```
bool("abc")
```

[43]:
```
True
```

For sequences, which we'll see in the next section, the Boolean representation is False for empty sequences and True for any other sequences

[44]:
```
bool([1, 2, 3])
```

[44]:
```
True
```

[45]:
```
bool([])
```

[45]:
```
False
```

4 Data Structures

We have seen Python's simple types: `int`, `float`, `complex`, `bool`, `str`, and so on. Python also has several built-in compound types, which act as containers for other types. These compound types are:

Type Name	Example	Description
`list`	`[1, 2, 3]`	Ordered collection
`tuple`	`(1, 2, 3)`	Immutable ordered collection
`dict`	`{'a':1, 'b':2, 'c':3}`	Unordered (key,value) mapping
`set`	`{1, 2, 3}`	Unordered collection of unique values

As you can see, round, square, and curly brackets have distinct meanings when it comes to the type of collection produced. We'll take a quick tour of these data structures here.

4.1 Lists

Lists are the basic *ordered* and *mutable* data collection type in Python. They can be defined with comma-separated values between square brackets; for example, here is a list of the first several prime numbers:

[46]:
```
L = [2, 3, 5, 7]
```

Lists have a number of useful properties and methods available to them. Here we'll take a quick look at some of the more common and useful ones:

[47]:
```
# Length of a list
len(L)
```

[47]:
```
4
```

[48]:
```
# Append a value to the end
L.append(11)
L
```

[48]:
```
[2, 3, 5, 7, 11]
```

[49]:
```
# Addition concatenates lists
L + [13, 17, 19]
```

[49]:
```
[2, 3, 5, 7, 11, 13, 17, 19]
```

[50]:
```
# sort() method sorts in-place
L = [2, 5, 1, 6, 3, 4]
L.sort()
L
```

[50]:
```
[1, 2, 3, 4, 5, 6]
```

In addition, there are many more built-in list methods; they are well-covered in Python's online documentation.

While we've been demonstrating lists containing values of a single type, one of the powerful features of Python's compound objects is that they can contain objects of *any* type, or even a mix of types. For example:

[51]:
```
L = [1, 'two', 3.14, [0, 3, 5]]
```

This flexibility is a consequence of Python's dynamic type system. Creating such a mixed sequence in a statically-typed language like C can be much more of a headache! We see that lists can even contain other lists as elements. Such type flexibility is an essential piece of what makes Python code relatively quick and easy to write.

So far we've been considering manipulations of lists as a whole; another essential piece is the accessing of individual elements. This is done in Python via *indexing* and *slicing*, which we'll explore next.

4.1.1 List indexing and slicing

Python provides access to elements in compound types through *indexing* for single elements, and *slicing* for multiple elements. As we'll see, both are indicated by a square-bracket syntax. Suppose we return to our list of the first several primes:

[52]:
```
L = [2, 3, 5, 7, 11]
```

Python uses *zero-based* indexing, so we can access the first and second element in using the following syntax:

[53]:
```
L[0]
```

[53]: 2

[54]:
```
L[1]
```

[54]: 3

Elements at the end of the list can be accessed with negative numbers, starting from -1:

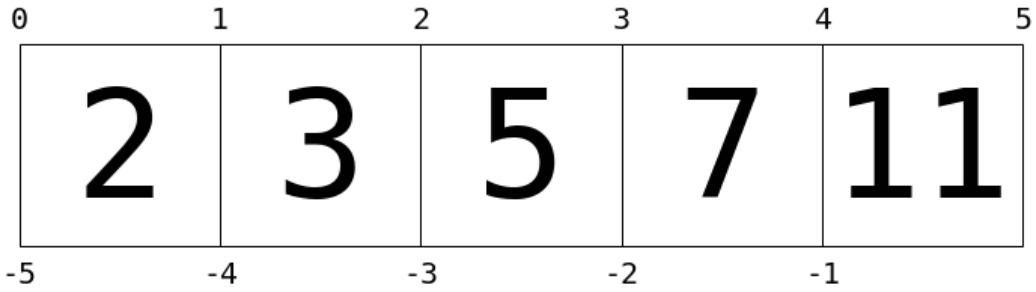
[55]:
```
L[-1]
```

[55]: 11

[56]:
```
L[-2]
```

[56]: 7

You can visualize this indexing scheme this way:



Here values in the list are represented by large numbers in the squares; list indices are represented by small numbers above and below. In this case, `L[2]` returns `5`, because that is the next value at index `2`.

Where *indexing* is a means of fetching a single value from the list, *slicing* is a means of accessing multiple values in sub-lists. It uses a colon to indicate the start point (inclusive) and end point (non-inclusive) of the sub-array. For example, to get the first three elements of the list, we can write:

[57]:
```
L[0:3]
```

[57]:
```
[2, 3, 5]
```

Notice where `0` and `3` lie in the preceding diagram, and how the slice takes just the values between the indices. If we leave out the first index, `0` is assumed, so we can equivalently write:

[58]:
```
L[:3]
```

[58]:
```
[2, 3, 5]
```

Similarly, if we leave out the last index, it defaults to the length of the list. Thus, the last three elements can be accessed as follows:

[59]:
```
L[-3:]
```

[59]:
```
[5, 7, 11]
```

Finally, it is possible to specify a third integer that represents the step size; for example, to select every second element of the list, we can write:

[60]:
```
L[::2]  # equivalent to L[0:len(L):2]
```

[60]:
```
[2, 5, 11]
```

A particularly useful version of this is to specify a negative step, which will reverse the array:

[61]:
```
L[::-1]
```

[61]:
```
[11, 7, 5, 3, 2]
```

Both indexing and slicing can be used to set elements as well as access them. The syntax is as you would expect:

[62]:
```
L[0] = 100
print(L)
```

```
[100, 3, 5, 7, 11]
```

[63]:
```
L[1:3] = [55, 56]
print(L)
```

```
[100, 55, 56, 7, 11]
```

A very similar slicing syntax is also used in many data science-oriented packages, including NumPy and Pandas (mentioned in the introduction).

Now that we have seen Python lists and how to access elements in ordered compound types, let's take a look at the other three standard compound data types mentioned earlier.

4.2 Tuples

Tuples are in many ways similar to lists, but they are defined with parentheses rather than square brackets:

[64]:
```
t = (1, 2, 3)
```

They can also be defined without any brackets at all:

[65]:
```
t = 1, 2, 3
print(t)
```

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

Like the lists discussed before, tuples have a length, and individual elements can be extracted using square-bracket indexing:

[66]:
```
len(t)
```

[66]: 3

[67]:
```
t[0]
```

[67]: 1

The main distinguishing feature of tuples is that they are *immutable*: this means that once they are created, their size and contents cannot be changed:

[68]:
```
t[1] = 4
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [68], line 1
----> 1 t[1] = 4

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

[69]:
```
t.append(4)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In [69], line 1
----> 1 t.append(4)

AttributeError: 'tuple' object has no attribute 'append'
```

Tuples are often used in a Python program; a particularly common case is in functions that have multiple return values. For example, the `as_integer_ratio()` method of floating-point objects returns a numerator and a denominator; this dual return value comes in the form of a tuple:

[70]:
```
x = 0.125
x.as_integer_ratio()
```

[70]: (1, 8)

These multiple return values can be individually assigned as follows:

[71]:
```
numerator, denominator = x.as_integer_ratio()
print(numerator / denominator)
```

```
0.125
```

The indexing and slicing logic covered earlier for lists works for tuples as well, along with a host of other methods. Refer to the online Python documentation for a more complete list of these.

4.3 Dictionaries

Dictionaries are extremely flexible mappings of keys to values, and form the basis of much of Python's internal implementation. They can be created via a comma-separated list of `key:value` pairs within curly braces:

[72]:
```
numbers = {'one':1, 'two':2, 'three':3}
```

Items are accessed and set via the indexing syntax used for lists and tuples, except here the index is not a zero-based order but valid key in the dictionary:

[73]:
```
# Access a value via the key
numbers['two']
```

[73]:
```
2
```

New items can be added to the dictionary using indexing as well:

[74]:
```
# Set a new key:value pair
numbers['ninety'] = 90
print(numbers)
```

```
{'one': 1, 'two': 2, 'three': 3, 'ninety': 90}
```

Keep in mind that dictionaries do not maintain any sense of order for the input parameters; this is by design. This lack of ordering allows dictionaries to be implemented very efficiently, so that random element access is very fast, regardless of the size of the dictionary (if you're curious how this works, read about the concept of a *hash table*). The python documentation has a complete list of the methods available for dictionaries.

4.4 Sets

The fourth basic collection is the set, which contains unordered collections of unique items. They are defined much like lists and tuples, except they use the curly brackets of dictionaries:

[75]:
```
primes = {2, 3, 5, 7}
odds = {1, 3, 5, 7, 9}
```

If you're familiar with the mathematics of sets, you'll be familiar with operations like the union, intersection, difference, symmetric difference, and others. Python's sets have all of these operations built-in, via methods or operators. For each, we'll show the two equivalent methods:

[76]:
```
# union: items appearing in either
primes | odds       # with an operator
primes.union(odds) # equivalently with a method
```

[76]:
```
{1, 2, 3, 5, 7, 9}
```

[77]:
```
# intersection: items appearing in both
primes & odds              # with an operator
primes.intersection(odds) # equivalently with a method
```

[77]:
```
{3, 5, 7}
```

[78]:
```
# difference: items in primes but not in odds
primes - odds           # with an operator
primes.difference(odds) # equivalently with a method
```

[78]:
```
{2}
```

[79]:
```
# symmetric difference: items appearing in only one set
primes ^ odds                      # with an operator
primes.symmetric_difference(odds) # equivalently with a method
```

[79]:
```
{1, 2, 9}
```

Many more set methods and operations are available. You've probably already guessed what I'll say next: refer to Python's online documentation for a complete reference.

5 Operators

In the previous section, we began to look at the semantics of Python variables and objects; here we'll dig into the semantics of the various *operators* included in the language. By the end of this section, you'll have the basic tools to begin comparing and operating on data in Python.

5.1 Arithmetic Operations

Python implements seven basic binary arithmetic operators, two of which can double as unary operators. They are summarized in the following table:

Operator	Name	Description
`a + b`	Addition	Sum of `a` and `b`
`a - b`	Subtraction	Difference of `a` and `b`
`a * b`	Multiplication	Product of `a` and `b`
`a / b`	True division	Quotient of `a` and `b`
`a // b`	Floor division	Quotient of `a` and `b`, removing fractional parts
`a % b`	Modulus	Integer remainder after division of `a` by `b`
`a ** b`	Exponentiation	`a` raised to the power of `b`
`-a`	Negation	The negative of `a`
`+a`	Unary plus	`a` unchanged (rarely used)

These operators can be used and combined in intuitive ways, using standard parentheses to group operations. For example:

[80]:
```
# addition, subtraction, multiplication
(4 + 8) * (6.5 - 3)
```

[80]:
```
42.0
```

Floor division is true division with fractional parts truncated:

[81]:
```
# True division
print(11 / 2)
```

```
5.5
```

[82]:
```
# Floor division
print(11 // 2)
```

```
5
```

The floor division operator was added in Python 3; you should be aware if working in Python 2 that the standard division operator (`/`) acts like floor division for integers and like true division for floating-point numbers.

Finally, I'll mention an eighth arithmetic operator that was added in Python 3.5: the `a @ b` operator, which is meant to indicate the *matrix product* of `a` and `b`, for use in various linear algebra packages.

5.2 Assignment Operations

We've seen that variables can be assigned with the "`=`" operator, and the values stored for later use. For example:

[83]:
```
a = 24
print(a)
```

```
24
```

We can use these variables in expressions with any of the operators mentioned earlier. For example, to add 2 to `a` we write:

[84]:
```
a + 2
```

[84]: 26

We might want to update the variable `a` with this new value; in this case, we could combine the addition and the assignment and write `a = a + 2`. Because this type of combined operation and assignment is so common, Python includes built-in update operators for all of the arithmetic operations:

[85]:
```
a += 2  # equivalent to a = a + 2
print(a)
```

```
26
```

There is an augmented assignment operator corresponding to each of the binary operators listed earlier; in brief, they are:

`a += b`	`a -= b`	`a *= b`	`a /= b`
`a //= b`	`a %= b`	`a **= b`	`a &= b`
`a |= b`	`a ^= b`	`a <<= b`	`a >>= b`

Each one is equivalent to the corresponding operation followed by assignment: that is, for any operator "■", the expression `a ■= b` is equivalent to `a = a ■ b`, with a slight catch. For mutable objects like lists, arrays, or DataFrames, these augmented assignment operations are actually subtly different than their more verbose counterparts: they modify the contents of the original object rather than creating a new object to store the result.

5.3 Comparison Operations

Another type of operation which can be very useful is comparison of different values. For this, Python implements standard comparison operators, which return Boolean values `True` and `False`. The comparison operations are listed in the following table:

Operation	Description	Operation	Description
`a == b`	`a` equal to `b`	`a != b`	`a` not equal to `b`
`a < b`	`a` less than `b`	`a > b`	`a` greater than `b`
`a <= b`	`a` less than or equal to `b`	`a >= b`	`a` greater than or equal to `b`

These comparison operators can be combined with the arithmetic and bitwise operators to express a virtually limitless range of tests for the numbers. For example, we can check if a number is odd by checking that the modulus with 2 returns 1:

[86]:
```
# 25 is odd
25 % 2 == 1
```

[86]:
```
True
```

[87]:
```
# 66 is odd
66 % 2 == 1
```

[87]:
```
False
```

We can string-together multiple comparisons to check more complicated relationships:

[88]:
```
# check if a is between 15 and 30
a = 25
15 < a < 30
```

[88]:
```
True
```

And, just to make your head hurt a bit, take a look at this comparison:

[89]:
```
-1 == ~0
```

[89]:
```
True
```

Recall that `~` is the bit-flip operator, and evidently when you flip all the bits of zero you end up with -1. If you're curious as to why this is, look up the *two's complement* integer encoding scheme, which is what Python uses to encode signed integers, and think about what happens when you start flipping all the bits of integers encoded this way.

5.4 Boolean Operations

When working with Boolean values, Python provides operators to combine the values using the standard concepts of "and", "or", and "not". Predictably, these operators are expressed using the words `and`, `or`, and `not`:

[90]:
```
x = 4
(x < 6) and (x > 2)
```

[90]:
```
True
```

[91]:
```
(x > 10) or (x % 2 == 0)
```

[91]:
```
True
```

[92]:
```
not (x < 6)
```

[92]:
```
False
```

Boolean algebra aficionados might notice that the XOR operator is not included; this can of course be constructed in several ways from a compound statement of the other operators. Otherwise, a clever trick you can use for XOR of Boolean values is the following:

[93]:
```
# (x > 1) xor (x < 10)
(x > 1) != (x < 10)
```

[93]:
```
False
```

These sorts of Boolean operations will become extremely useful when we begin discussing *control flow statements* such as conditionals and loops.

One sometimes confusing thing about the language is when to use Boolean operators (`and`, `or`, `not`), and when to use bitwise operations (`&`, `|`, `~`). The answer lies in their names: Boolean operators should be used when you want to compute *Boolean values (i.e., truth or falsehood) of entire statements*. Bitwise operations should be used when you want to *operate on individual bits or components of the objects in question.*

5.5 Identity and Membership Operators

Like `and`, `or`, and `not`, Python also contains prose-like operators to check for identity and membership. They are the following:

Operator	Description
`a is b`	True if `a` and `b` are identical objects
`a is not b`	True if `a` and `b` are not identical objects
`a in b`	True if `a` is a member of `b`
`a not in b`	True if `a` is not a member of `b`

5.5.1 Identity Operators: "`is`" and "`is not`"

The identity operators, "`is`" and "`is not`" check for *object identity*. Object identity is different than equality, as we can see here:

[94]:
```
a = [1, 2, 3]
b = [1, 2, 3]
```

[95]:
```
a == b
```

[95]:
```
True
```

[96]:
```
a is b
```

[96]:
```
False
```

[97]:
```
a is not b
```

[97]:
```
True
```

What do identical objects look like? Here is an example:

[98]:
```
a = [1, 2, 3]
b = a
a is b
```

[98]:
```
True
```

The difference between the two cases here is that in the first, `a` and `b` point to *different objects*, while in the second they point to the *same object*. As we saw in the previous section, Python variables are pointers. The "`is`" operator checks whether the two variables are pointing to the same container (object), rather than referring to what the container contains. With this in mind, in most cases that a beginner is tempted to use "`is`" what they really mean is `==`.

5.5.2 Membership operators

Membership operators check for membership within compound objects. So, for example, we can write:

[99]:
```
1 in [1, 2, 3]
```

[99]:
```
True
```

[100]:
```
2 not in [1, 2, 3]
```

[100]:
```
False
```

These membership operations are an example of what makes Python so easy to use compared to lower-level languages such as C. In C, membership would generally be determined by manually constructing a loop over the list and checking for equality of each value. In Python, you just type what you want to know, in a manner reminiscent of straightforward English prose.

6 Control Flow

Control flow is where the rubber really meets the road in programming. Without it, a program is simply a list of statements that are sequentially executed. With control flow, you can execute certain code blocks conditionally and/or repeatedly: these basic building blocks can be combined to create surprisingly sophisticated programs!

Here we'll cover *conditional statements* (including "`if`", "`elif`", and "`else`"), *loop statements* (including "`for`" and "`while`" and the accompanying "`break`", "`continue`", and "`pass`").

6.1 Conditional Statements: `if-elif-else`:

Conditional statements, often referred to as *if-then* statements, allow the programmer to execute certain pieces of code depending on some Boolean condition. A basic example of a Python conditional statement is this:

[101]:
```
x = -15

if x == 0:
    print(x, "is zero")
elif x > 0:
    print(x, "is positive")
elif x < 0:
    print(x, "is negative")
else:
    print(x, "is unlike anything I've ever seen...")
```

```
-15 is negative
```

Note especially the use of colons (`:`) and whitespace to denote separate blocks of code.

Python adopts the `if` and `else` often used in other languages; its more unique keyword is `elif`, a contraction of "else if". In these conditional clauses, `elif` and `else` blocks are optional; additionally, you can optinally include as few or as many `elif` statements as you would like.

6.2 `for` loops

Loops in Python are a way to repeatedly execute some code statement. So, for example, if we'd like to print each of the items in a list, we can use a `for` loop:

[102]:
```
for N in [2, 3, 5, 7]:
    print(N, end=' ') # print all on same line
```

```
2 3 5 7
```

Notice the simplicity of the `for` loop: we specify the variable we want to use, the sequence we want to loop over, and use the "`in`" operator to link them together in an intuitive and readable way. More precisely, the object to the right of the "`in`" can be any Python *iterator*. An iterator can be thought of as a generalized sequence.

For example, one of the most commonly-used iterators in Python is the `range` object, which generates a sequence of numbers:

[103]:
```
for i in range(10):
    print(i, end=' ')
```

```
0 1 2 3 4 5 6 7 8 9
```

Note that the range starts at zero by default, and that by convention the top of the range is not included in the output. Range objects can also have more complicated values:

[104]:
```
# range from 5 to 10
list(range(5, 10))
```

[104]:
```
[5, 6, 7, 8, 9]
```

[105]:
```
# range from 0 to 10 by 2
list(range(0, 10, 2))
```

[105]:
```
[0, 2, 4, 6, 8]
```

You might notice that the meaning of `range` arguments is very similar to the slicing syntax that we covered in Lists.

Note that the behavior of `range()` is one of the differences between Python 2 and Python 3: in Python 2, `range()` produces a list, while in Python 3, `range()` produces an iterable object.

6.3 `while` loops

The other type of loop in Python is a `while` loop, which iterates until some condition is met:

[106]:
```
i = 0
while i < 10:
    print(i, end=' ')
    i += 1
```

```
0 1 2 3 4 5 6 7 8 9
```

The argument of the `while` loop is evaluated as a boolean statement, and the loop is executed until the statement evaluates to False.

6.4 `break` and `continue`: Fine-Tuning Your Loops

There are two useful statements that can be used within loops to fine-tune how they are executed:

- The `break` statement breaks-out of the loop entirely
- The `continue` statement skips the remainder of the current loop, and goes to the next iteration

These can be used in both `for` and `while` loops.

Here is an example of using `continue` to print a string of odd numbers. In this case, the result could be accomplished just as well with an `if-else` statement, but sometimes the `continue` statement can be a more convenient way to express the idea you have in mind:

[107]:
```
for n in range(20):
    # if the remainder of n / 2 is 0, skip the rest of the loop
    if n % 2 == 0:
        continue
    print(n, end=' ')
```

```
1 3 5 7 9 11 13 15 17 19
```

Here is an example of a `break` statement used for a less trivial task. This loop will fill a list with all Fibonacci numbers up to a certain value:

[108]:
```
a, b = 0, 1
amax = 100
L = []

while True:
    (a, b) = (b, a + b)
    if a > amax:
        break
    L.append(a)

print(L)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Notice that we use a `while True` loop, which will loop forever unless we have a break statement!

7 Functions

So far, our scripts have been simple, single-use code blocks. One way to organize our Python code and to make it more readable and reusable is to factor-out useful pieces into reusable *functions*.

7.1 Using Functions

Functions are groups of code that have a name, and can be called using parentheses. We've seen functions before. For example, `print` in Python 3 is a function:

[109]:
```
print('abc')
```

```
abc
```

Here `print` is the function name, and `'abc'` is the function's *argument*.

In addition to arguments, there are *keyword arguments* that are specified by name. One available keyword argument for the `print()` function (in Python 3) is `sep`, which tells what character or characters should be used to separate multiple items:

[110]:
```
print(1, 2, 3)
```

```
1 2 3
```

[111]:
```
print(1, 2, 3, sep='--')
```

```
1--2--3
```

When non-keyword arguments are used together with keyword arguments, the keyword arguments must come at the end.

7.2 Defining Functions

Functions become even more useful when we begin to define our own, organizing functionality to be used in multiple places. In Python, functions are defined with the `def` statement. For example, we can encapsulate a version of our Fibonacci sequence code from the previous section as follows:

[112]:
```
def fibonacci(N):
    L = []
    a, b = 0, 1
    while len(L) < N:
        a, b = b, a + b
        L.append(a)
    return L
```

Now we have a function named `fibonacci` which takes a single argument `N`, does something with this argument, and `returns` a value; in this case, a list of the first `N` Fibonacci numbers:

[113]:
```
fibonacci(10)
```

[113]:
```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

If you're familiar with strongly-typed languages like `C`, you'll immediately notice that there is no type information associated with the function inputs or outputs. Python functions can return any Python object, simple or compound, which means constructs that may be difficult in other languages are straightforward in Python.

For example, multiple return values are simply put in a tuple, which is indicated by commas:

[114]:
```
def real_imag_conj(val):
    return val.real, val.imag, val.conjugate()

r, i, c = real_imag_conj(3 + 4j)
print(r, i, c)
```

```
3.0 4.0 (3-4j)
```

7.3 Default Argument Values

Often when defining a function, there are certain values that we want the function to use *most* of the time, but we'd also like to give the user some flexibility. In this case, we can use *default values* for arguments. Consider the `fibonacci` function from before. What if we would like the user to be able to play with the starting values? We could do that as follows:

[115]:
```
def fibonacci(N, a=0, b=1):
    L = []
    while len(L) < N:
        a, b = b, a + b
        L.append(a)
    return L
```

With a single argument, the result of the function call is identical to before:

[116]:
```
fibonacci(10)
```

[116]:
```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

But now we can use the function to explore new things, such as the effect of new starting values:

[117]:
```
fibonacci(10, 0, 2)
```

[117]:
```
[2, 2, 4, 6, 10, 16, 26, 42, 68, 110]
```

The values can also be specified by name if desired, in which case the order of the named values does not matter:

[118]:
```
fibonacci(10, b=3, a=1)
```

[118]:
```
[3, 4, 7, 11, 18, 29, 47, 76, 123, 199]
```

7.4 `*args` and `**kwargs`: Flexible Arguments

Sometimes you might wish to write a function in which you don't initially know how many arguments the user will pass. In this case, you can use the special form `*args` and `**kwargs` to catch all arguments that are passed. Here is an example:

[119]:
```
def catch_all(*args, **kwargs):
    print("args =", args)
    print("kwargs = ", kwargs)
```

[120]:
```
catch_all(1, 2, 3, a=4, b=5)
```

```
args = (1, 2, 3)
kwargs =  {'a': 4, 'b': 5}
```

[121]:
```
catch_all('a', keyword=2)
```

```
args = ('a',)
kwargs =  {'keyword': 2}
```

Here it is not the names `args` and `kwargs` that are important, but the `*` characters preceding them. `args` and `kwargs` are just the variable names often used by convention, short for "arguments" and "keyword arguments". The operative difference is the asterisk characters: a single `*` before a variable means "expand this as a sequence", while a double `**` before a variable means "expand this as a dictionary". In fact, this syntax can be used not only with the function definition, but with the function call as well!

[122]:
```
inputs = (1, 2, 3)
keywords = {'pi': 3.14}

catch_all(*inputs, **keywords)
```

```
args = (1, 2, 3)
kwargs =  {'pi': 3.14}
```