2

Simple Data Types

We’ll look at a number of data types which are built-in as well as some that are part of Python’s standard library. We’ll start with Python’s numeric types. This includes three built-in types – int, float, and complex – plus standard library types Fraction and Decimal.

We’ll also look at strings, str, and simple collections, tuple. These are more complex than numbers because they contain multiple items. Because their behavior is less complex than the kinds of objects we’ll see in later chapters, they serve as a good introduction to the general concept of Sequences in Python.

Note the capitalization of the names of Fraction and Decimal. The built-in type names start with a lower-case letter. Types that we must import will have a module name that starts with a lower case letter, but the type name will start with a capital letter. This convention is widespread but not completely universal.

All of the types we’ll look at in this chapter will have the common feature of immutability. When looking at a simple integer, we expect immutability: 355 is always 355; it can’t magically change state to become 113. This concept applies to the two collections we’ll look at, also. Once built, a string or a tuple cannot be changed.

Some popular languages (e.g., Java, C++) have primitive types which are not proper objects – they’re not instances of a class – which means that they escape from the rules that apply to objects. This is emphatically not the case for Python. All Python objects are proper instances of a class.

In this chapter, we’ll look at the built-in functions for converting to and from string representations. This will help us when displaying output or converting input from a string to a useful Python object.

Note that we’re continuing to play fast and loose with formal Python syntax. We’ll defer a detailed examination of the syntax rules until in Chapter 3, “Expressions and Output”. For now, the kinds of simple expression statements we’re focusing on must be complete on a single line.

Introducing the built-in operators

Before looking at the various kinds of numbers available, we’ll introduce the Python operators. The operators fall into three broad groups:

* Arithmetic: + - \* \*\* / // %
* Bit-Oriented: << >> & | ^ ~
* Comparison: < > <= >= == !=

The distinctions among these groups are partly subjective. There’s only a small technical distinction among these operators. Most of the operators are binary, one (~) is only unary sense, and a few (+, -, \*, \*\*) can be used in either context.

The +, -, \*, /, and % operators have meanings similar to other programming languages. There is a unary arithmetic meaning for – and +, also. Python adds the \*\* operator for raising a number to a power. The \*\* operator is higher precedence than unary -; this means that -2\*\*4 is -16.

The bit-oriented operators apply only to integers. They also, it turns out, apply to sets. These are emphatically not logical operators. The logical operators are described in Chapter 5, “Logic, Comparisons, Conditions”.

Doing comparisons

The comparison operators, < > == != <= >=, have meanings similar to other programming languages. The coercion rules apply to comparison among numbers. If the objects are of mixed types, one of them will be coerced “up” the numeric tower from integer → float or float → complex. The result of a comparison is a boolean – True or False – irrespective of the types of the two operands.

The various coercion rules do not apply to strings or other objects. Strings are not implicitly converted to numbers. 2 != '2' is true because integer 2 is not string '2'.

We’ll look at comparisons in more detail in Chapter 5, “Logic, Comparisons, and Conditions.”

Using integer numbers

Python integers are objects of the class int. These objects have the largest number of operators, including all of the arithmetic, bit-oriented and comparison operators.

Integer values are limited by available memory. This means they can be quite large. We can easily compute 1000!, a number with over 2,500 digits. We’ll save the details for Chapter 8, “More Advanced Functions.” A number of similarly gargantuan size is .

>>> 2\*\*8530

610749...581824

It’s a very large number. We’ve elided most of it. It’s easily represented in Python.

Generally, we provide integer literals in decimal, base 10. We can also write literals in three other bases: hexadecimal, octal and binary.

The prefix of 0x is the prefix for base 16 values: 0x10 is 16. We can use letters a-f as is typical in many other programming languages; 0xdeadbeef is valid. The prefix 0o (zero and the letter oh) is used for base eight; try to avoid using the maliciously confusing 0O (zero and capital oh) for octal values. For example, 0o33653337357. We can write base two literal values using the 0b prefix: 0b10 is 2.

Using the bit-oriented operators

The bit-oriented operators are defined for integers. They’re not defined for complex or floating-point objects.

The << and >> operators perform bit shifting. 1 << 8, for example, is 256. We’ve shifted the value 1 to the left 8 bit positions.

The &, |, and ^ operators compute the bit-wise “and”, bit-wise “or”, and bit-wise “xor” of two integer values. Here are some examples:

>>> 9 & 5

1

>>> 9 | 5

13

>>> 9 ^ 3

10

To visualize these operators, we can use the bin() function to see the binary values involved.

>>> bin(9)

'0b1001'

>>> bin(5)

'0b101'

Using the bin() function can clarify how the bits of 9|5 combine to create the bits of 13.

The ~ operator is the bit-wise two’s complement of an integer value. ~14, for example, is -15.

These are emphatically not logical operators. The logical operators are described in Chapter 5, “Logic, Comparisons, Conditions”.

Do not confuse a & b with a and b.

a & b computes a bit-wise “and” of the bits in the integers a and b.

a and b computes the boolean “and” based on the truth values of a and b.

Using rational numbers

Rational numbers are fractions composed of two integer values. Python doesn’t have a built-in rational number type. We must import the Fraction class using this:

>>> from fractions import Fraction

This will introduce the Fraction class definition to our global environment. Once we have this, we can create objects of class Fraction as follows:

>>> Fraction(355,113)

Fraction(355, 113)

The arithmetic and comparison operators apply to fractions. When doing mixed-type expressions, fractions fit into the numeric tower above integers and below floating-point values. Here’s an example of an integer coerced to a fraction:

>>> Fraction(4,2)\*3

Fraction(6, 1)

To perform an operation that involves Fraction value and an int value means that the int object must be coerced up to the Fraction class.

We can extract the numerator and denominator of a fraction using their attribute names. Here’s an example:

>>> a= Fraction(355,113)\*5

>>> a.numerator

1775

>>> a.denominator

113

We’ve created a Fraction object, a, from an expression involving a Fraction object and an integer. We’ve then extracted the numerator and denominator attributes of the variable a.

Using decimal numbers

For currency calculations, we generally use Decimal numbers. Python doesn’t have a built-in decimal number type. We import the Decimal class using this:

>>> from decimal import Decimal

This will introduce the Decimal class definition to our global environment. We can now create Decimal objects. It’s important to avoid accidentally mixing Decimal and float values, because float values are only an approximation. To be sure that Decimal values are exact, we must use only integers or strings.

>>> Decimal("2.72")

Decimal('2.72')

We’ve created a Decimal value from a string. The resulting Decimal object will represent this exactly, carefully preserving appropriate decimal places and performing round-off as required.

When we look at common financial calculations, Decimal is required. Here’s an example:

>>> (Decimal('512.97')+Decimal('5.97'))\*Decimal('0.075')

Decimal('38.92050')

We’ve added two prices, $512.97 and $5.97 and computed a sales tax of 7.5%. The tax is exactly $38.92050. This is generally rounded to $38.92.

If we try this kind of financial calculation with floating-point values, we have a bit of a problem:

>>> (512.97+5.97)\*0.075

38.920500000000004

The floating-point approximations don’t produce an exact answer.

The Python coercion rules work nicely between Decimal and int values. We can do Decimal('3.99')\*3 and get Decimal('11.97') as the exact answer.

The coercion rules aren’t implemented by the Decimal and float classes. It might make some sense for Decimal values to be coerced up to float values. On the other hand, this might indicate a profound programming error when mixing exact currency values and floating-point approximations. Since this is ambiguous, and debatable, the general approach followed by Python is summarized by this line from Tim Peters’ the Zen of Python:

In the face of ambiguity, refuse the temptation to guess.

Consequently, mixing Decimal and float leads to TypeError exceptions instead of coercion up the numeric tower and the resulting switch from exact to approximate values. We must explicitly convert Decimal to float to do mixed-type expressions.

Using floating-point numbers

Floating-point values are instances of the class float. These objects work with the arithmetic and comparison operators. They don’t participate in the bit-oriented operators.

The details of Python floating-point implementations can vary. CPython depends on the standard C libraries, which should provide reasonably consistent results on a wide variety of hardware and OS platforms. The C libraries generally use IEEE 754 floating-point values; Python’s float type is the C language double. This means that a float will be a 64-bit value with (effectively) a 53-bit fraction and an 11-bit exponent. The exponent ranges is from to .

We can write floating-point numbers two ways: as digits with a decimal point, as well as “scientific” notation:

>>> 6335.437

6335.437

>>> 6.335437E3

6335.437

The “E” notation shows a power of 10. That means 6.335437E3 is .

It’s very important to note that floating-point values are an approximation. We can’t emphasize enough that they’re not exact and should not be used for currency calculations.

Here’s an example of what happens when working with floating-point approximations:

>>> (5\*\*6)\*\*(1/6)

4.999999999999999

This should not be astonishing in any way. Mathematically, . Since values like 1/6 don’t have exact binary representations, this kind of expression reveals the consequences of working with approximations.

The fact that floating-point numbers use a binary representation leads to interesting complications. A number like 1/6 has no exact decimal representation; we can say .1666... to indicate that the decimal positions repeat infinitely. A number like 1/5 has an exact decimal representation, 0.2. Neither of these numbers has an exact binary representation. Since we must use a finite number of bits, we’ll notice slight discrepancies between idealized values and the finite values produced on a digital computer.

Note that exact equality comparisons between floating-point numbers, while permitted, is generally not the best idea. In Chapter 5, “Logic, Comparisons, and Conditions,” we’ll address the proper way to use a narrow range instead of exact equality. Instead of a == b, we need to focus on abs(a-b) < ε.

Using complex numbers

The top of Python’s number tower is the complex type. These can be thought of as expressions built from a pair of floating-point numbers: one is a real value, the other is an imaginary value. The imaginary value is multiplied by . We write (2+3j) to mean .

When working with complex numbers, we’ll often import the cmath library instead of the math library. The math.sqrt() function is constrained to work only with float values, and will raise an exception rather than provide an imaginary value. The cmath.sqrt() function will provide a proper imaginary value if required.

This library shows us that is essentially true:

>>> cmath.e\*\*(cmath.pi\*1j)+1

1.2246467991473532e-16j

Note that we used 1j to represent . If we try to use the identifier j – without a number in front of it – it is seen as a simple a variable. The value 1j is a complex literal because it starts with a digit and ends with j.

Since floating-point values have about 53 bits, which is about 16 decimal digits, we can expect float approximations of irrational values like π and e to be off by about .

The numeric tower

We’ve seen Python’s three built-in numeric types – int, float, complex – plus two more types – Fraction, Decimal – imported from the standard library. The numbers module in the standard library provides four base class definitions for the numeric types. We rarely need to use this module explicitly; it’s a formality that we need when we have to implement our own numeric types.

The numeric types form a kind of “tower” that parallels the various kinds of numbers seen in conventional mathematics. The foundation of the tower is integers. Rational numbers are above integers. Floating-point values are still further up, and complex numbers are at the top of the tower.

A common expectation is that a language will automatically coerce numeric values to permit expressions like 2\*2.718 to work properly and produce a useful result. When multiplying an integer and a float value, we expect integers to be coerced to a floating-point value.

In order for this to work, there are two general rules for the result of a binary arithmetic operation:

* If both operands are the same type, the result has that type. For example, 2 \*\* 1024 does not produce a floating-point result. It produces an immense integer.
* If the operands are mixed, one of them will be coerced “up” the numeric tower from integer → rational → floating-point → complex.

There is one notable exception to the above rules. The / and // operators define two different kinds of division. The / operator provides true division: even integer operands will yield a floating-point result. For example:

>>> 355/113

3.1415929203539825

The // operator provides floor division: the result will be truncated as if it were integer-only division. The resulting type won’t be coerced, but the answer will be truncated. For example:

>>> 355./113.

3.1415929203539825

>>> 355.//113.

3.0

The presence of the // operator means that an expression which is designed with integers in mind will also work correctly with floating-point values. Similarly, we may write an expression with an informal expectation of floating-point values; by using /, it will also work with integers.

Note that these coercion rules among numeric types do not apply to strings or other objects. Strings are not implicitly converted to numbers. The expression '2'+2 results in a TypeError exception. We’ll look at the explicit conversions below in the “Using the built-in conversion functions” section.

The tower metaphor provides a handy way to remember the coercion rules. Given two values from different levels, the lower-level value is coerced up the tower to the higher-level values.

The math libraries

The Python library has six modules relevant for mathematical work. These are described in chapter nine of the Python Standard Library document, “Numeric and Mathematical Modules.” Beyond this, we have external libraries like numpy (http://www.numpy.org) and scipy (http://www.scipy.org). These libraries include vast collections of sophisticated algorithms. For an even more sophisticated toolset, the Anaconda project (https://store.continuum.io/cshop/anaconda/) combines numpy, scipy and 18 more packages.

These are the relevant built-in math packages:

* numbers. This module defines the essential numeric abstractions. We rarely need this unless we’re going to invent an entirely new kind of number.
* math. This module has a large collection of functions. It includes basic sqrt(), the various trigonometric functions (sine, cosine, etc.) and the various log-related functions. It has functions for working with the internals of floating-point numbers. It also has the gamma function and the error function.
* cmath. This module is the complex version of the math library. We use the cmath library so that we can seamlessly move between float and complex values.
* decimal. Import the Decimal class from this module to work with currency values accurately.
* fractions. Import the Fraction class to work with a precise rational fraction value.
* random. This module contains the essential random number generator. It has a number of other functions to produce random values in various ranges or with various constraints. For example random.gauss() produces a gaussian, normal distribution of floating-point values.

There are three common styles for importing from these libraries. Here is a summary of these styles:

* import random. We’ll use this when we want to be perfectly explicit about the origin of a name elsewhere in our code. We’ll be writing code similar to random.gauss() and random.randint() using the module name as an explicit qualifier.
* from random import gauss, randint. This introduces two selected names from the random module into the global namespace. We can use gauss() and randint() without a qualifying module name.
* from random import \*. This will introduce all of the available names in the random module as globals in our application. This can be helpful for exploring and experimenting at the >>> prompt. This may not be appropriate in a larger program because it can introduce a large number of irrelevant names.

A less-commonly used feature allows us to rename objects brought in via the import statement. We might want to use from cmath import sqrt as csqrt to rename the cmath.sqrt() function to csqrt(). We have to be careful to avoid ambiguity and confusion when using this import-as renaming feature.

Using bits and boolean values

As noted above, the bit-oriented operators & | ^ and ~ have nothing to do with Python’s actual boolean operators and, or, not, and if-else. We’ll look at boolean values, logic operators and related programming in Chapter 5, “Logic, Comparisons, Conditions”.

If we misuse the bit-oriented operators & or | in place of a logical and or or, things may appear very peculiar:

>>> 5 > 6 & 3 > 1

True

>>> (5 > 6) & (3 > 1)

False

The first example is clearly wrong. Why? This is because the & operator has relatively high priority. It’s not a logical connective, it’s more like an arithmetic operator. The & operator is performed first: 6&3 evaluates to 2. Given this, the resulting expression, 5 > 2 > 1, is True.

When we group the comparisons to perform them first, we’ll get a False for 5>6, and a True for 3>1. When we applying the & operator the result will be False, which is what we expected. Using bit operators improperly as logical connectives can work if we use parenthesis heavily to be sure that the bit operators are performed last. It’s a very bad idea, however.

It’s easier, more clear, and higher performance to use the proper boolean operators shown in Chapter 5, “Logic, Comparisons, Conditions”.

Working with sequences

In this chapter, we’ll introduce Python sequence collections. We’ll look at strings and tuples as the first two examples of this class. Python offers a number of other sequence collections; we’ll look at them in Chapter 6, “More Complex Data Types”. All of these sequences have some common features.

Python sequences identify the individual elements by position. Positions numbers start with zero. Here’s a tuple with five elements:

>>> t=("hello", 3.14, 23, None, True)

>>> t[0]

'hello'

>>> t[4]

True

In addition to the expected ascending numbers, Python offers reverse numbering, also. Position -1 is the end of the sequence:

>>> t[-1]

True

>>> t[-2]

>>> t[-5]

'hello'

Note that position 3 (or -2) has a value of None. The REPL doesn’t display the None object, so the value of t[-2] appears to be missing. For more visible evidence that this value is None, use this

>>> t[3] is None

True

The sequences use an extra comparison operator, **in**. We can ask if a given value occurs in a collection:

>>> "hello" in t

True

>>> 2.718 in t

False

Slicing and dicing a sequence

We can extract a subsequence, called a slice, from a sequence using more complex subscript expressions. Here’s a substring of a longer string:

>>> "multifaceted"[5:10]

'facet'

The [5:10] expression is a slice which starts at position 5 and extends to the position before 10. Python generally relies on “half-open” intervals. The starting position of a slice is included, the stop position is excluded.

We can omit the starting position from a slice, writing [:pos]. If the start value of a slice is omitted, it’s 0. We can omit the ending, also, writing it [pos:]. If the stop value of a slice is omitted, it’s the length of the sequence, given by the len() function.

The way that Python uses these half-open intervals means that we can partition a string with very tidy syntax:

>>> "multifaceted"[:5]

'multi'

>>> "multifaceted"[5:]

'faceted'

In this example, we’ve taken the first five characters in the first slice. We’ve taken everything after the first five characters in the second slice. Because the numbers are both five, we can be completely sure that the entire string is accounted for.

And yes, we can omit both values from the slice: "word"[:] will create a copy of the entire string. This is an odd but sometimes useful construct for duplicating an object.

There’s a third parameter to a slice. We generally call the positions “start”, “stop”, and “step”. The step size is 1 by default. We can use a form like "abcdefg"[::2] to provide an explicit step, and pick characters in positions 0, 2, 4, and 6. The form "abcdefg"[1::2] will pick the odd positions: 1, 3, and 5.

The step size can be negative, also. This will enumerate the index values in reverse order. The value of "word"[::-1] is 'drow'.

Using string and bytes values

Python string values are similar – in some respects – to simple numeric types. There are a few arithmetic-like operators available and all of the comparisons are defined. Strings are immutable: we cannot change a string. We can, however, easily build new strings from existing strings, making the mutability question as irrelevant for string objects as it is for number objects.

Python has two kinds of string values:

* Unicode. These strings use the entire Unicode character set. These are the default strings Python uses. The input-output libraries are all capable of a wide variety of Unicode encoding and decoding. The name for this type is str. It’s a built-in type, so it starts with a lower-case letter.
* Bytes. Many file formats and network protocols are defined over bytes, not Unicode characters. Python uses ASCII encoding for bytes. Special arrangements must be made to process bytes. The internal type name is bytes.

We can trivially encode Unicode into a sequence of bytes. We can just as easily decode a sequence of bytes to discover the Unicode characters. We’ll show these two methods in the “Converting between Unicode and bytes” section, after we’ve looked at literals and operators.

Writing string literals

String literals are characters surrounded by string delimiters. Python offers a variety of string delimiters to solve a variety of problems. The most common literals will create Unicode strings:

* **Short String**. Use either " or ' to surround the string. For example: "Don't Touch" has an embedded apostrophe. 'Speak "friend" and enter' has embedded quotes. In the rare case where we have both, we can use \ to escape a quote: '"Don\'t touch," he said.' uses apostrophe as delimiters, and an escaped apostrophe within the string. While a string literal must be complete on a single line, a ‘\n’ will expand into a proper newline character internally.
* **Long String**. Use either """ or ''' to surround a multi-line string. The string can span as many lines as necessary. A long string can include any characters imaginable except for the terminating triple-quote or triple-apostrophe.

Python has a moderate number of \ escape sequences to allow us to enter characters that aren’t present on our keyboards. If we use ordinary str literals, Python replaces all the escape sequences with proper Unicode characters. In an ordinary bytes literal, the escape sequence becomes a one-byte ASCII character.

Many Python programs are saved as pure ASCII text, but this is not a requirement. When saving a file in ASCII, escapes will be required for non-ASCII Unicode characters. When saving files in a Unicode encoding, then relatively few escapes are required, since any Unicode character available on our keyboard can be entered directly.

Here are two examples of the same string:

>>> "String with π×r²"

>>> "String with \u03c0\u00d7r\N{superscript two}"

The first string uses Unicode characters; the file must be saved in some appropriate encoding like UTF-8 for this to work. The second string uses escape sequences to describe the Unicode characters. The \u sequence is followed by a four-digit hex value. The \N{...} escape allows writing the name of the character. A \U escape – not shown in the example – requires an 8-digit hex value. The second example can be saved in any encoding, including ASCII.

The most commonly-used escape sequences are \", \', \n, \t, and \\ to create a quote inside a quoted string, an apostrophe inside an apostrophe delimited string, a newline, a tab, and a \ character. There are a few others, but their meanings are so obscure that numeric codes usually make more sense. For example, \v, should probably be written as \x0b or \u000b; the original meaning behind \v is largely lost to history.

Note that '\u000b' is the actual character. We also have '\u240b' which is a Unicode glyph, '␋', that symbolizes that vertical tab character. Most of the non-printing ASCII control characters also have these symbolic glyphs.

Using raw string literals

Sometimes, we need to provide strings in which the \ is not an escape character. When preparing regular expressions, for example, we prefer not be forced to write \\ to represent a single \. Similarly, when working with Windows file names, we don’t want "C:\temp" to have an ASCII horizontal tab character ('\u0008') replace the ‘\t’ sequence of characters in the middle of the string literal. We could write "C:\\temp" but it seems error-prone.

To avoid this escape processing, Python offers the raw string. We can prefix any of the previous four flavors of delimiters with the letter r or R. For example, r'\b[a-zA-Z\_]\w+\b', is a raw string. The \’s will be left intact by Python: the ‘\b’ sequences are not translated to ‘\u0008’ characters.

If we do this without using the r" as the raw string delimiter, we’ll be creating a string literal equivalent to this: '\x08[a-zA-Z\_]\\w+\x08'. This shows how a ‘\b’ characters can be transformed to ‘\x08’ in a non-raw string. Omitting the leading r' leads to a string that does not represent the regular expression we intended.

Using byte string literals

We may also need to include byte strings in our programs as well as Unicode strings. In order to do this, we’ll use a prefix of b or B in front of the string delimiter. A byte string is limited to ASCII characters and escape sequences that produce single-byte ASCII characters.

Generally, byte strings will focus on the hexadecimal escape, \xhh, with two hex digits for byte strings. We can also use the octal escape, \odd, with octal digits.

We can also prepare raw byte strings using any combination of r or R paired with b or B as a prefix to the string. Here’s a regular expression in ASCII bytes:

>>> rb"\\x[0-9a-fA-F]+"

b'\\\\x[0-9a-fA-F]+'

The output is in a canonical notation using lengthy escapes for the ‘\\’ regular expression pattern.

To be fastidious, we are also able to use a u" prefix to indicate that a given string is explicitly Unicode. This is relatively rare because it restates the default assumption. It can come in handy in a program where byte strings predominate; the use of u"some string" can make the Unicode literal stand out from numerous b"bytes" literals.

Using the string operators

Two of the arithmetic operators, + and \*, are defined for both classes of string objects, str and bytes. We can used the + operator to concatenate two string objects, creating a longer string. Interesting, we can use the \* operator to multiply a string and an integer to create a longer string: "="\*3 is '==='.

Additionally, adjacent string literals are combined into a larger string during code parsing. Here’s an example:

>>> "adjacent " 'literals'

'adjacent literals'

Since this happens at parse time, it only works for string literals. For variables or other expressions, there must be a proper + operator.

All of the comparison operators work for strings. The comparison operators compare two strings character by character. We’ll look at this in detail in Chapter 5, “Logic, Comparisons, and Conditions.”

We cannot use string operators with mixed types of operands. Trying to do "hello" + b"world" will raise a TypeError exception. We must either encode the Unicode str into bytes, or decode the bytes into a Unicode str object.

Strings are sequence collections. We can extract characters and slices from the. Strings also work with the in operator. We can ask if a particular character or a substring occurs in a string like this:

>>> "i" in "bankrupted"

False

>>> "bank" in "bankrupted"

True

The first example shows the typical use for the in operator: checking to see if a given item is in the collection. This use of in applies to many other kinds of collections. The second example shows a feature that is unique to strings: we’re looking for a given substring in a longer string.

Converting between Unicode and bytes

Most of the Python I/O libraries are aware of OS file encodings. When working with text files, we rarely need to explicitly provide an encoding. We’ll examine the details of Python’s input-output capabilities in Chapter 10, “Files, Databases, Networks, and Contexts”.

When we need to encode Unicode characters as a string of bytes, we’ll use the encode() method of a string. Here’s an example:

>>> 'String with π×r²'.encode("utf-8")

b'String with \xcf\x80\xc3\x97r\xc2\xb2'

We’ve provided a literal Unicode string, and encoded this into UTF-8 bytes. Python has numerous encoding schemes, all defined in the codecs module.

To decode the Unicode string represented by a string of bytes, we use the decode() method of the bytes. Here’s an example:

>>> b'very \xe2\x98\xba\xef\xb8\x8e'.decode('utf-8')

'very ☺︎'

We’ve provided a byte string with eleven individually hex-encoded bytes. We decoded this include six Unicode characters.

Note that there are several aliases for the supported encodings. We’ve used “utf-8” and “UTF-8”. There are still more shown in codecs chapter of the Python Standard Library.

Of these, the ASCII codec is frequently necessary. In addition to ASCII, many strings and text files are encoded in UTF-8. When downloading data on the internet, there’s often a header or other indicator that provides the encoding in the rare case that it’s not UTF-8.

In some cases, we’ll have a document which is bytes, written in traditional ASCII. To work with ASCII files, we’ll convert the bytes – using the ASCII encoding – to Unicode characters. Similarly, we can encode a subset of Unicode characters using the ASCII encoding instead of UTF-8.

It’s possible that a given sequences bytes does not properly encode Unicode characters. This may be because the wrong encoding is being used to decode the bytes. Or it could be because the bytes are incorrect. The decode() method has additional parameters to define what to do when the bytes cannot be decoded. The values for the errors argument are strings:

* "strict" means that exceptions are raised. This is the default.
* "ignore" means that invalid bytes will be skipped.
* "replace" means that a default character will be inserted. This is defined in the codecs module. The '\ufffd' character is the default replacement.

The choice of error handling is highly application-specific.

Using string methods

A string object has a large number of method functions. Most of these apply both to str and bytes objects. These can be separated into four groups:

* transformers, which create new strings from old strings,
* creators, which create a string from non-string object(s),
* accessors, which access a string and return a fact about that string,
* parsers, which examine a string and decompose the string, or create new data objects from the string.

The transformer group of method functions includes capitalize(), center(), expandtabs(), ljust(), lower(), rjust(), swapcase(), title(), upper(), and zfill(). These methods all make general changes to the characters of a string to create a transformed result. Methods like lower() and upper() are used frequently to normalize case for comparisons:

>>> "WoRd".lower()

'word'

Using this technique allows us to write programs which are more tolerant of character strings with minor errors.

Additional transformers include functions like strip(), rstrip(), lstrip(), and replace(). The functions in the strip family remove whitespace. It’s common to use rstrip() on input lines to remove any trailing spaces and the trailing newline character which might be present.

The replace() function will replace any substring with another substring. If we want to do multiple independent replacements, we can do something like this.

>>> "$12,345.00".replace("$","").replace(",","")

'12345.00'

This will create an intermediate string with the “$” removed. It will create a second intermediate string from that with the “,” removed. This kind of processing is handy for cleaning up raw data.

Accessing details of a string

We use accessor methods to determine facts about the string; the results may be boolean or integer values. For example, the count() method returns a count of the number of places an argument substring or character was found in the object string.

Some widely-used methods include the find(), rfind(), index(), and rindex() methods which will find the position of a substring in the object string. The find() methods return a special value of -1 if the substring isn’t found. The index() methods raise a ValueError exception if the substring isn’t found. The “r” versions find the right-most occurrence of the target substring. All of these methods are available for both str and bytes objects.

The endswith() and startswith() methods are boolean functions; they examine the beginning or ending of a string. Here are some examples:

>>> "pleonastic".endswith("tic")

True

>>> "rediscount".find("disc")

2

>>> "postlaunch".find("not")

-1

The first example shows how we can check the ending of a string with the endswith() method. The second example shows how the find() method locates the offset of a given substring in a longer string. The third example shows show the find() method returns a signal value of -1 if the substring can’t be found.

Additionally, there are seven boolean pattern-matching functions. These are isalnum(), isalpha(), isdigit(), islower(), isspace(), istitle(), and isupper(). These will return True if the function matches a given pattern. For example, "13210".isdigit() is True.

Parsing strings into substrings

There are a few method functions which we can use to decompose a string into substrings. We’ll hold off on looking at split(), join(), and partition() in detail until Chapter 3, “Expressions and Output”.

As a quick overview, we’ll note that split() will split a string into a sequence of strings based on the locating a possibly repeating separator substring. We might use an expression like '01.03.05.15'.split('.') to create the sequence ['01', '03', '05', '15'] from the longer string, by splitting on the ‘.’ character. The join() method is the inverse of split(). That means that "-".join(['01', '03', '05', '15']) will create a new string from the individual strings and the separator; the result is '01-03-05-15'. Partition can be viewed as a single-item split to separate the head of a string from the tail.

Python’s assignment statement deals very gracefully with a method like this which returns more than one value. In Chapter 4, “Variables, Assignment and Scoping Rules” we’ll look at multiple assignment closely.

The split() method should not be used to parse file names, nor should the join() method be used to build file names. There’s a separate module, os.path, which handles this properly by applying OS-specific rules.

Using the tuple collection

The tuple is one of the simplest collections available in Python. It is one of the many kinds of Python sequences. A tuple has a fixed number of items. For example, we might work with (x,y) coordinates or (r,g,b) colors. In these cases, the number of elements in each tuple is fixed by the problem domain. We don’t want the flexibility of a collection that can vary in length.

Generally, we’ll include ()’s around a tuple to set it off from surrounding syntax. These ()’s aren’t always required; Python creates tuple objects implicitly in some common contexts. However, they’re always a good idea. If we write an assignment statement like this:

a = 2, 3

This statement will implicitly create a 2-tuple – (2, 3) – and assign the object to the variable a.

The tuple class is part of Python’s family of Sequence classes; we can extract the items of a tuple using their positional indices. The str and byte classes are also examples of Sequence. In addition to simple index values, we can use slice notation to pick items from a tuple.

The value () is a zero-length tuple. To create a singleton tuple, we must use ()’s and include a ,: this means that (12,) is a singleton tuple. If we omit the , we’ve written an expression, not a singleton tuple.

A trailing comma is required for a singleton tuple. An extra comma at the end of a tuple is quietly ignored everywhere else: (1, 1, 2) is equal to (1, 1, 2,).

The tuple class offers only two method functions: count() and index(). We can count the number of occurrences of a given item in a tuple, and we can locate the position of an item in a tuple.

The None object

One very simple kind of Python object is the None object. It has few methods, and there’s only a single instance of this object available. It is a handy way to identify something as missing or not applicable. It’s often used as a default value for optional parameters to a function.

The None object is a Singleton; there can be only one. This object is immutable, also: we can’t change it in any way.

In interactive use of Python, the REPL doesn’t print the None object. For example, when we evaluate the print() function, the proper result of this function is always None. The side-effect of this function is to print things on our console. Looking forward to Chapter 3, “Expressions and Output,” we’ll show this quick example of a function which returns None:

>>> a = print("hello world")

hello world

>>> a

>>> a is None

True

We’ve evaluated the print() function and saved the result of the print function in the a variable. The visible side-effect of printing is to see the string value displayed on the console. The result is the None object, which is not printed. We can, however, use the is comparison operator to see that the value of a really is the None object.

The consequences of immutability

Python has two broad flavors of objects: mutable and immutable. A mutable object has an internal state that can be updated through operators or method functions. An immutable object’s state cannot be changed.

The canonical examples of immutable objects are the numbers. The number 2 must always have a single, immutable value midway between 1 and 3. We can’t change the state of 2 to make it 3 without making a mockery of the idea of mathematical truth.

In the chapter 6, “More Complex Data Types”, we’ll look at a number of mutable data structures. The most important three mutable collections are the set, list, and dict. These objects can have items added, and removed; we can change the state of the object.

In addition to numbers being immutable, three other common structures are also immutable: str, bytes, and tuple. Because strings and bytes are immutable, the string manipulation methods will always create a new string object from one or more existing string objects.

This means we cannot mutate characters or substrings within a longer string. We might think we need to attempt something like this:

>>> word="vokalizers"

>>> word[2]= "c"

It can’t work because a string object is immutable. We always build new strings from the old string’s parts. We do it like this:

>>> word= word[:2]+"c"+word[3:]

This works by extracting pieces of the original string and including new characters mixed with the old.

Using the built-in conversion functions

We have a number of conversion functions between the various types of data we’ve seen in this chapter. Each of the built-in numeric types has a proper constructor function. As with many Python functions, each of these has a number of different kinds of arguments it can handle:

* int(). Creates an int from a wide variety of other objects.
	+ int(3.718) for another number,
	+ int('48879') for a string in base 10,
	+ int('beef', 16) for a string in the given base – 16 in this example.
	+ Plus, the int() function can gracefully ignore the extra prefix characters on numbers written in Python literal syntax: int('0b1010',2), int('0xbeef',16), and int('0o123',8).
* float(). Creates a float from other objects.
	+ float(7331) for another number,
	+ float('4.8879e5') for a decimal string.
* complex(). Creates complex values from a variety of objects.
	+ complex(23) creates (23+0j).
	+ complex(23, 3) creates (23+3j).
	+ complex('23+2j') creates (23+2j).

We can convert single numbers, pairs of numbers, and even some strings into Fraction objects:

* Fraction(2,3). This is the most common way to create Fraction objects.
* Fraction(2.718) This creates a value Fraction(765048986699563, 281474976710656). This shows how floating-point values are actually approximations. If we wanted a more accurate value, we should do a meaningful conversion ourselves, using Fraction(2718,1000), which would avoid the error bits present in many floating-point values.
* Fraction("3/4"). This also works very nicely to create a proper Fraction object.

When we convert a float value to a Fraction, the results look unusual. However, considering that float values are an approximation, the Fraction value reveals the nature of the approximation.

We can also convert integers, strings, and floats to Decimal objects:

* Decimal(2). Interestingly, this produces Decimal('2') as the result. This shows us that the preferred format for Decimal values is strings.
* Decimal('2.718') will produce the expected value. This is generally how we create Decimal objects.
* Decimal(2.718) will produce a value that reflects floating-point approximations. Decimal('2.717999999999999971578290569595992565155029296875'). Because of this, we generally avoid creating Decimal objects from float objects.

We have a number of additional conversions from numbers to various kinds of strings: bin(), oct(), hex(), and str() produce strings in base 2, 8, 16, and 10 respectively. We can also use various formatting features of numbers using "{0:b}".format(x) for binary, "{0:o}".format(x) for octal, "{0:x}".format(x) for hexadecimal. If we include the “#” modifier in the format string, we have considerable flexibility in the strings produced. For example:

>>> "{0:x}".format(12)

'c'

>>> "{0:#x}".format(12)

'0xc'

These functions show a considerable breadth of ways to create numbers from strings and create formatted strings from numbers.

Summary and Looking Forward

We’ve looked at some core data types available in Python. We’ve looked at five different kinds of numbers, including integers, floating-point, complex, Fraction and Decimal. Each fills a different niche. Three of these a built-in, the other two must be imported from the standard library.

We’ve also looked at three different kinds of collections. The tuple is a simple sequence of items with relatively few methods. The str is a Unicode string, which has a broad variety of methods to create new strings as transformations of existing strings. The bytes is a byte string, which also has a variety of methods. We can decode bytes to create Unicode strings. We can encode Unicode strings into bytes.

We’ve touched on how the import statement is used to introduce new types and new modules. We’ll add features from the standard library.

We’ve also looked at a number of functions to convert between various numeric types. Many of these functions will also convert strings to numbers. We’ll make heavy use of int() and float() to convert strings to numbers. The reverse – converting numbers to strings – can be done with the str() function. It can be done better, however, with the formatting tools we’ll look at in the next chapter.

In Chapter 3, “Expressions and Output,” we’ll build on these foundational concepts. We’ll look in more depth at Python language syntax. We’ll also look at functions for creating nicely formatted output. This will allow us to write simple programs. In Chapter 4, “Variables, Assignment, and Scoping Rules,” we’ll add even more of the essential language features so that we can write more sophisticated programs.