Chapter 1. Rapid Introduction to Python’s “Beautiful Heart”

Python’s “Beautiful Heart”

In this section we will learn about eight key pieces of Python, and in the next section we will show how these pieces can be used to write a couple of small but realistic programs. There is much more to say about all of the things covered in this section, so if as you read it you feel that Python is missing something or that things are sometimes done in a long-winded way, peek ahead using the forward references or using the table of contents or index, and you will almost certainly find that Python has the feature you want and often has more concise forms of expression than we show here—and a lot more besides.

Piece #1: Data Types

One fundamental thing that any programming language must be able to do is represent items of data. Python provides several built-in data types, but we will concern ourselves with only two of them for now. Python represents integers (positive and negative whole numbers) using the \texttt{int} type, and it represents strings (sequences of Unicode characters) using the \texttt{str} type. Here are some examples of integer and string literals:

\begin{verbatim}
-973
210624583337114373395836055367340864637790190801098222508621955072
0
"Infinitely Demanding"
'Simon Critchley'
'positively αβγ€÷©'
'
\end{verbatim}

Incidentally, the second number shown is $2^{217}$—the size of Python’s integers is limited only by machine memory, not by a fixed number of bytes. Strings can be delimited by double or single quotes, as long as the same kind are used at both ends, and since Python uses Unicode, strings are not limited to ASCII characters, as the penultimate string shows. An empty string is simply one with nothing between the delimiters.

Python uses square brackets ([]) to access an item from a sequence such as a string. For example, if we are in a Python Shell (either in the interactive interpreter, or in IDLE) we can enter the following—the Python Shell’s output is shown in \texttt{lightface}; what you type is shown in \texttt{bold}:
Traditionally, Python Shells use >>> as their prompt, although this can be changed. The square brackets syntax can be used with data items of any data type that is a sequence, such as strings and lists. This consistency of syntax is one of the reasons that Python is so beautiful. Note that all Python index positions start at 0.

In Python, both str and the basic numeric types such as int are immutable—that is, once set, their value cannot be changed. At first this appears to be a rather strange limitation, but Python’s syntax means that this is a non-issue in practice. The only reason for mentioning it is that although we can use square brackets to retrieve the character at a given index position in a string, we cannot use them to set a new character. (Note that in Python a character is simply a string of length 1.)

To convert a data item from one type to another we can use the syntax datatype(item). For example:

```python
>>> int("45")
45
>>> str(912)
'912'
```

The int() conversion is tolerant of leading and trailing whitespace, so int(" 45 ") would have worked just as well. The str() conversion can be applied to almost any data item. We can easily make our own custom data types support str() conversion, and also int() or other conversions if they make sense, as we will see in Chapter 6. If a conversion fails, an exception is raised—we briefly introduce exception-handling in Piece #5, and fully cover exceptions in Chapter 4.

Strings and integers are fully covered in Chapter 2, along with other built-in data types and some data types from Python’s standard library. That chapter also covers operations that can be applied to immutable sequences, such as strings.

**Piece #2: Object References**

Once we have some data types, the next thing we need are variables in which to store them. Python doesn’t have variables as such, but instead has object references. When it comes to immutable objects like int and str, there is no discernable difference between a variable and an object reference. As for mutable objects, there is a difference, but it rarely matters in practice. We will use the terms variable and object reference interchangeably.

*Shallow and deep copying* 146
Let’s look at a few tiny examples, and then discuss some of the details.

```python
x = "blue"
y = "green"
z = x
```

The syntax is simply `objectReference = value`. There is no need for predeclaration and no need to specify the value’s type. When Python executes the first statement it creates a `str` object with the text “blue”, and creates an object reference called `x` that refers to the `str` object. For all practical purposes we can say that “variable `x` has been assigned the ‘blue’ string”. The second statement is similar. The third statement creates a new object reference called `z` and sets it to refer to the same object that the `x` object reference refers to (in this case the `str` containing the text “blue”).

The `=` operator is not the same as the variable assignment operator in some other languages. The `=` operator binds an object reference to an object in memory. If the object reference already exists, it is simply re-bound to refer to the object on the right of the `=` operator; if the object reference does not exist it is created by the `=` operator.

Let’s continue with the `x`, `y`, `z` example, and do some rebinding—as noted earlier, comments begin with a `#` and continue until the end of the line:

```python
print(x, y, z) # prints: blue green blue
z = y
print(x, y, z) # prints: blue green green
x = z
print(x, y, z) # prints: green green green
```

After the fourth statement (`x = z`), all three object references are referring to the same `str`. Since there are no more object references to the “blue” string, Python is free to garbage-collect it.

*Figure 1.2* shows the relationship between objects and object references schematically.

*Figure 1.2. Object references and objects*
The names used for object references (called identifiers) have a few restrictions. In particular, they may not be the same as any of Python’s keywords, and must start with a letter or an underscore and be followed by zero or more nonwhitespace letter, underscore, or digit characters. There is no length limit, and the letters and digits are those defined by Unicode, that is, they include, but are not limited to, ASCII’s letters and digits (“a”, “b”, ..., “z”, “A”, “B”, ..., “Z”, “0”, “1”, ..., “9”). Python identifiers are case-sensitive, so for example, LIMIT, Limit, and limit are three different identifiers. Further details and some slightly exotic examples are given in Chapter 2.

Identifiers and keywords 51

Python uses dynamic typing, which means that an object reference can be rebound to refer to a different object (which may be of a different data type) at any time. Languages that use strong typing (such as C++ and Java) allow only those operations that are defined for the data types involved to be performed. Python also applies this constraint, but it isn’t called strong typing in Python’s case because the valid operations can change—for example, if an object reference is rebound to an object of a different data type. For example:

```
route = 866
print(route, type(route)) # prints: 866 <class 'int'>
route = "North"
print(route, type(route)) # prints: North <class 'str'>
```

Here we create a new object reference called `route` and set it to refer to a new `int` of value 866. At this point we could use `/` with `route` since division is a valid operation for integers. Then we reuse the `route` object reference to refer to a new `str` of value “North”, and the `int` object is scheduled for garbage collection since now no object reference refers to it. At this point using `/` with `route` would cause a `TypeError` to be raised since `/` is not a valid operation for a string.

The `type()` function returns the data type (also known as the “class”) of the data item it is given—this function can be very useful for testing and debugging, but would not normally appear in production code, since there is a better alternative as we will see in Chapter 6.

`isinstance()` 242

If we are experimenting with Python code inside the interactive interpreter or in a Python Shell such as the one provided by IDLE, simply typing the name of an object reference is enough to have Python print its value. For example:

```
>>> x = "blue"
>>> y = "green"
>>> z = x
```
>>> x
'blue'
>>> x, y, z
('blue', 'green', 'blue')

This is much more convenient than having to call the `print()` function all the time, but works only when using Python interactively—any programs and modules that we write must use `print()` or similar functions to produce output. Notice that Python displayed the last output in parentheses separated by commas—this signifies a tuple, that is, an ordered immutable sequence of objects. We will cover tuples in the next piece.

**Piece #3: Collection Data Types**

It is often convenient to hold entire collections of data items. Python provides several collection data types that can hold items, including associative arrays and sets. But here we will introduce just two: tuple and list. Python tuples and lists can be used to hold any number of data items of any data types. Tuples are immutable, so once they are created we cannot change them. Lists are mutable, so we can easily insert items and remove items whenever we want.

Tuples are created using commas (,), as these examples show—and note that here, and from now on, we don’t use bold to distinguish what you type:

```python
>>> "Denmark", "Finland", "Norway", "Sweden"
('Denmark', 'Finland', 'Norway', 'Sweden')
>>> "one",
('one',)
```

When Python outputs a tuple it encloses it in parentheses. Many programmers emulate this and always enclose the tuple literals they write in parentheses. If we have a one-item tuple and want to use parentheses, we must still use the comma—for example, (1,). An empty tuple is created by using empty parentheses, (). The comma is also used to separate arguments in function calls, so if we want to pass a tuple literal as an argument we must enclose it in parentheses to avoid confusion.

**tuple type** 108

**Creating and calling functions** 36

Here are some example lists:

```
[1, 4, 9, 16, 25, 36, 49]
['alpha', 'bravo', 'charlie', 'delta', 'echo']
```
One way to create a list is to use square brackets ([ ]) as we have done here; later on we will see other ways. The fourth list shown is an empty list.

Under the hood, lists and tuples don’t store data items at all, but rather object references. When lists and tuples are created (and when items are inserted in the case of lists), they take copies of the object references they are given. In the case of literal items such as integers or strings, an object of the appropriate data type is created in memory and suitably initialized, and then an object reference referring to the object is created, and it is this object reference that is put in the list or tuple.

Like everything else in Python, collection data types are objects, so we can nest collection data types inside other collection data types, for example, to create lists of lists, without formality. In some situations the fact that lists, tuples, and most of Python’s other collection data types hold object references rather than objects makes a difference—this is covered in Chapter 3.

In procedural programming we call functions and often pass in data items as arguments. For example, we have already seen the print() function. Another frequently used Python function is len(), which takes a single data item as its argument and returns the “length” of the item as an int. Here are a few calls to len():

```python
>>> len("one",)
1
>>> len([3, 5, 1, 2, "pause", 5])
6
>>> len("automatically")
13
```

Tuples, lists, and strings are “sized”, that is, they are data types that have a notion of size, and data items of any such data type can be meaningfully passed to the len() function. (An exception is raised if a nonsized data item is passed to len().)
All Python data items are objects (also called instances) of a particular data type (also called a class). We will use the terms data type and class interchangeably. One key difference between an object, and the plain items of data that some other languages provide (e.g., C++ or Java’s built-in numeric types), is that an object can have methods. Essentially, a method is simply a function that is called for a particular object. For example, the list type has an append() method, so we can append an object to a list like this:

```python
>>> x = ['zebra', 49, -879, 'aardvark', 200]
>>> x.append('more')
>>> x
['zebra', 49, -879, 'aardvark', 200, 'more']
```

The x object knows that it is a list (all Python objects know what their own data type is), so we don’t need to specify the data type explicitly. In the implementation of the append() method the first argument will be the x object itself—this is done automatically by Python as part of its syntactic support for methods.

The append() method mutates, that is, changes, the original list. This is possible because lists are mutable. It is also potentially more efficient than creating a new list with the original items and the extra item and then rebinding the object reference to the new list, particularly for very long lists.

In a procedural language the same thing could be achieved by using the list’s append() like this (which is perfectly valid Python syntax):

```python
>>> list.append(x, "extra")
>>> x
['zebra', 49, -879, 'aardvark', 200, 'more', 'extra']
```

Here we specify the data type and the data type’s method, and give as the first argument the data item of the data type we want to call the method on, followed by any additional arguments. (In the face of inheritance there is a subtle semantic difference between the two syntaxes; the first form is the one that is most commonly used in practice. Inheritance is covered in Chapter 6.)

If you are unfamiliar with object-oriented programming this may seem a bit strange at first. For now, just accept that Python has conventional functions called like this:
```python
functionName(arguments); and methods which are called like this:
objectName.methodName(arguments). (Object-oriented programming is covered in Chapter 6.)
```

The dot (“access attribute”) operator is used to access an object’s attributes. An attribute can be any kind of object, although so far we have shown only method attributes. Since an attribute can be an object that has attributes, which in turn can have attributes, and so on, we can use as many dot operators as necessary to access the particular attribute we want.
The list type has many other methods, including `insert()` which is used to insert an item at a given index position, and `remove()` which removes an item at a given index position. As noted earlier, Python indexes are always 0-based.

We saw before that we can get characters from strings using the square brackets operator, and noted at the time that this operator could be used with any sequence. Lists are sequences, so we can do things like this:

```python
>>> x
['zebra', 49, -879, 'aardvark', 200, 'more', 'extra']
>>> x[0]
'zebra'
>>> x[4]
200
```

Tuples are also sequences, so if `x` had been a tuple we could retrieve items using square brackets in exactly the same way as we have done for the `x` list. But since lists are mutable (unlike strings and tuples which are immutable), we can also use the square brackets operator to set list elements. For example:

```python
>>> x[1] = "forty nine"
>>> x
['zebra', 'forty nine', -879, 'aardvark', 200, 'more', 'extra']
```

If we give an index position that is out of range, an exception will be raised—we briefly introduce exception-handling in Piece #5, and fully cover exceptions in Chapter 4.

We have used the term sequence a few times now, relying on an informal understanding of its meaning, and will continue to do so for the time being. However, Python defines precisely what features a sequence must support, and similarly defines what features a sized object must support, and so on for various other categories that a data type might belong to, as we will see in Chapter 8.

Lists, tuples, and Python’s other built-in collection data types are covered in Chapter 3.

**Piece #4: Logical Operations**

One of the fundamental features of any programming language is its logical operations. Python provides four sets of logical operations, and we will review the fundamentals of all of them here.

**The Identity Operator**

Since all Python variables are really object references, it sometimes makes sense to ask whether two or more object references are referring to the same object. The `is` operator is a binary operator that returns `True` if its left-hand object reference is referring to the same object as its right-hand object reference. Here are some examples:
Note that it usually does not make sense to use `is` for comparing `int`, `str`, and most other data types since we almost invariably want to compare their values. In fact, using `is` to compare data items can lead to unintuitive results, as we can see in the preceding example, where although \( a \) and \( b \) are initially set to the same list values, the lists themselves are held as separate `list` objects and so `is` returns `False` the first time we use it.

One benefit of identity comparisons is that they are very fast. This is because the objects referred to do not have to be examined themselves. The `is` operator needs to compare only the memory addresses of the objects—the same address means the same object.

The most common use case for `is` is to compare a data item with the built-in null object, `None`, which is often used as a place-marking value to signify “unknown” or “nonexistent”:

```
>>> a = "Something"
>>> b = None
>>> a is not None, b is None
(True, True)
```

To invert the identity test we use `is not`.

The purpose of the identity operator is to see whether two object references refer to the same object, or to see whether an object is `None`. If we want to compare object values we should use a comparison operator instead.

**Comparison Operators**

Python provides the standard set of binary comparison operators, with the expected semantics: `<` less than, `<=` less than or equal to, `==` equal to, `!=` not equal to, `>=` greater than or equal to, and `>` greater than. These operators compare object values, that is, the objects that the object references used in the comparison refer to. Here are a few examples typed into a Python Shell:

```
>>> a = 2
>>> b = 6
>>> a == b
False
>>> a < b
True
>>> a <= b, a != b, a >= b, a > b
(True, True, False, False)
```
Everything is as we would expect with integers. Similarly, strings appear to compare properly too:

```python
>>> a = "many paths"
>>> b = "many paths"
>>> a is b
False
>>> a == b
True
```

Although `a` and `b` are different objects (have different identities), they have the same values, so they compare equal. Be aware, though, that because Python uses Unicode for representing strings, comparing strings that contain non-ASCII characters can be a lot subtler and more complicated than it might at first appear—we will fully discuss this issue in Chapter 2.

Comparing strings 68

In some cases, comparing the identity of two strings or numbers—for example, using `a is b`—will return `True`, even if each has been assigned separately as we did here. This is because some implementations of Python will reuse the same object (since the value is the same and is immutable) for the sake of efficiency. The moral of this is to use `==` and `!=` when comparing values, and to use `is` and `is not` only when comparing with `None` or when we really do want to see if two object references, rather than their values, are the same.

One particularly nice feature of Python’s comparison operators is that they can be chained. For example:

```python
>>> a = 9
>>> 0 <= a <= 10
True
```

This is a nicer way of testing that a given data item is in range than having to do two separate comparisons joined by logical `and`, as most other languages require. It also has the additional virtue of evaluating the data item only once (since it appears once only in the expression), something that could make a difference if computing the data item’s value is expensive, or if accessing the data item causes side effects.

Thanks to the “strong” aspect of Python’s dynamic typing, comparisons that don’t make sense will cause an exception to be raised. For example:

```python
>>> "three" < 4
Traceback (most recent call last):
  ...
TypeError: unorderable types: str() < int()
```
When an exception is raised and not handled, Python outputs a traceback along with the exception’s error message. For clarity, we have omitted the traceback part of the output, replacing it with an ellipsis. The same TypeError exception would occur if we wrote "3" < 4 because Python does not try to guess our intentions—the right approach is either to explicitly convert, for example, \texttt{int("3") < 4}, or to use comparable types, that is, both integers or both strings.

[*] A traceback (sometimes called a backtrace) is a list of all the calls made from the point where the unhandled exception occurred back to the top of the call stack.

Dealing with runtime errors 415

Python makes it easy for us to create custom data types that will integrate nicely so that, for example, we could create our own custom numeric type which would be able to participate in comparisons with the built-in int type, and with other built-in or custom numeric types, but not with strings or other non-numeric types.

Alternative FuzzyBool 256

The Membership Operator

For data types that are sequences or collections such as strings, lists, and tuples, we can test for membership using the \texttt{in} operator, and for nonmembership using the \texttt{not in} operator. For example:

```python
>>> p = (4, "frog", 9, -33, 9, 2)
>>> 2 in p
True
>>> "dog" not in p
True
```

For lists and tuples, the \texttt{in} operator uses a linear search which can be slow for very large collections (tens of thousands of items or more). On the other hand, \texttt{in} is very fast when used on a dictionary or a set; both of these collection data types are covered in Chapter 3. Here is how \texttt{in} can be used with a string:

```python
>>> phrase = "Wild Swans by Jung Chang"
>>> "J" in phrase
True
>>> "han" in phrase
```
Conveniently, in the case of strings, the membership operator can be used to test for substrings of any length. (As noted earlier, a character is just a string of length 1.)

**Logical Operators**

Python provides three logical operators: `and`, `or`, and `not`. Both `and` and `or` use short-circuit logic and return the operand that determined the result—they do not return a Boolean (unless they actually have Boolean operands). Let’s see what this means in practice:

```python
>>> five = 5
>>> two = 2
>>> zero = 0
>>> five and two
2
>>> two and five
5
>>> five and zero
0
```

If the expression occurs in a Boolean context, the result is evaluated as a Boolean, so the preceding expressions would come out as `True`, `True`, and `False` in, say, an `if` statement.

```python
>>> nought = 0
>>> five or two
5
>>> two or five
2
>>> zero or five
5
>>> zero or nought
0
```

The `or` operator is similar; here the results in a Boolean context would be `True`, `True`, `True`, and `False`.

The `not` unary operator evaluates its argument in a Boolean context and always returns a Boolean result, so to continue the earlier example, `not (zero or nought)` would produce `True`, and `not two` would produce `False`.

**Piece #5: Control Flow Statements**

We mentioned earlier that each statement encountered in a `.py` file is executed in turn, starting with the first one and progressing line by line. The flow of control can be diverted by a function or method call or by a control structure, such as a conditional branch or a loop statement. Control is also diverted when an exception is raised.
In this subsection we will look at Python’s if statement and its while and for loops, deferring consideration of functions to Piece #8, and methods to Chapter 6. We will also look at the very basics of exception-handling; we cover the subject fully in Chapter 4. But first we will clarify a couple of items of terminology.

A Boolean expression is anything that can be evaluated to produce a Boolean value (True or False). In Python, such an expression evaluates to False if it is the predefined constant False, the special object None, an empty sequence or collection (e.g., an empty string, list, or tuple), or a numeric data item of value 0; anything else is considered to be True. When we create our own custom data types (e.g., in Chapter 6), we can decide for ourselves what they should return in a Boolean context.

In Python-speak a block of code, that is, a sequence of one or more statements, is called a suite. Because some of Python’s syntax requires that a suite be present, Python provides the keyword pass which is a statement that does nothing and that can be used where a suite is required (or where we want to indicate that we have considered a particular case) but where no processing is necessary.

The if Statement

The general syntax for Python’s if statement is this:[*]

```python
if boolean_expression1:
    suite1
elif boolean_expression2:
    suite2
...
elif boolean_expressionN:
    suiteN
else:
    else_suite
```

[*] In this book, ellipses (...) are used to indicate lines that are not shown.

There can be zero or more elif clauses, and the final else clause is optional. If we want to account for a particular case, but want to do nothing if it occurs, we can use pass as that branch’s suite.

The first thing that stands out to programmers used to C++ or Java is that there are no parentheses and no braces. The other thing to notice is the colon: This is part of the syntax and is easy to forget at first. Colons are used with else, elif, and essentially in any other place where a suite is to follow.

Unlike most other programming languages, Python uses indentation to signify its block structure. Some programmers don’t like this, especially before they have tried it, and some get quite emotional about the issue. But it takes just a few days to get used to, and after a few weeks or months, brace-free code seems much nicer and less cluttered to read than code that uses braces.
Since suites are indicated using indentation, the question that naturally arises is, “What kind of indentation?” The Python style guidelines recommend four spaces per level of indentation, and only spaces (no tabs). Most modern text editors can be set up to handle this automatically (IDLE’s editor does of course, and so do most other Python-aware editors). Python will work fine with any number of spaces or with tabs or with a mixture of both, providing that the indentation used is consistent. In this book, we follow the official Python guidelines.

Here is a very simple if statement example:

```python
if x:
    print("x is nonzero")
```

In this case, if the condition (x) evaluates to True, the suite (the print() function call) will be executed.

```python
if lines < 1000:
    print("small")
elif lines < 10000:
    print("medium")
else:
    print("large")
```

This is a slightly more elaborate if statement that prints a word that describes the value of the lines variable.

**The while Statement**

The while statement is used to execute a suite zero or more times, the number of times depending on the state of the while loop’s Boolean expression. Here’s the syntax:

```python
while boolean_expression:
    suite
```

Actually, the while loop’s full syntax is more sophisticated than this, since both break and continue are supported, and also an optional else clause that we will discuss in Chapter 4. The break statement switches control to the statement following the innermost loop in which the break statement appears—that is, it breaks out of the loop. The continue statement switches control to the start of the loop. Both break and continue are normally used inside if statements to conditionally change a loop’s behavior.

```python
while True:
    item = get_next_item()
    if not item:
        break
    process_item(item)
```
This while loop has a very typical structure and runs as long as there are items to process. (Both get_next_item() and process_item() are assumed to be custom functions defined elsewhere.) In this example, the while statement’s suite contains an if statement, which itself has a suite—as it must—in this case consisting of a single break statement.

The for ... in Statement

Python’s for loop reuses the in keyword (which in other contexts is the membership operator), and has the following syntax:

```python
for variable in iterable:
    suite
```

Just like the while loop, the for loop supports both break and continue, and also has an optional else clause. The variable is set to refer to each object in the iterable in turn. An iterable is any data type that can be iterated over, and includes strings (where the iteration is character by character), lists, tuples, and Python’s other collection data types.

```python
for country in ["Denmark", "Finland", "Norway", "Sweden"]:
    print(country)
```

Here we take a very simplistic approach to printing a list of countries. In practice it is much more common to use a variable:

```python
countries = ["Denmark", "Finland", "Norway", "Sweden"]
for country in countries:
    print(country)
```

In fact, an entire list (or tuple) can be printed using the print() function directly, for example, print(countries), but we often prefer to print collections using a for loop (or a list comprehension, covered later), to achieve full control over the formatting.

List comprehensions

```python
for letter in "ABCDEFGHIJKLMNOPQRSTUVWXYZ":
    if letter in "AEIOU":
        print(letter, "is a vowel")
    else:
        print(letter, "is a consonant")
```

In this snippet the first use of the in keyword is part of a for statement, with the variable letter taking on the values "A", "B", and so on up to "Z", changing at each iteration of the loop. On the snippet’s second line we use in again, but this time as the membership testing operator. Notice
also that this example shows nested suites. The for loop’s suite is the if...else statement, and both the if and the else branches have their own suites.

Basic Exception Handling

Many of Python’s functions and methods indicate errors or other important events by raising an exception. An exception is an object like any other Python object, and when converted to a string (e.g., when printed), the exception produces a message text. A simple form of the syntax for exception handlers is this:

```python
try:
    try_suite
except exception1 as variable1:
    exception_suite1
...
except exceptionN as variableN:
    exception_suiteN
```

Note that the as variable part is optional; we may care only that a particular exception was raised and not be interested in its message text.

The full syntax is more sophisticated; for example, each except clause can handle multiple exceptions, and there is an optional else clause. All of this is covered in Chapter 4.

The logic works like this. If the statements in the try block’s suite all execute without raising an exception, the except blocks are skipped. If an exception is raised inside the try block, control is immediately passed to the suite corresponding to the first matching exception—this means that any statements in the suite that follow the one that caused the exception will not be executed. If this occurs and if the as variable part is given, then inside the exception-handling suite, variable refers to the exception object.

If an exception occurs in the handling except block, or if an exception is raised that does not match any of the except blocks in the first place, Python looks for a matching except block in the next enclosing scope. The search for a suitable exception handler works outward in scope and up the call stack until either a match is found and the exception is handled, or no match is found, in which case the program terminates with an unhandled exception. In the case of an unhandled exception, Python prints a traceback as well as the exception’s message text.

Dealing with runtime errors  415

Here is an example:

```python
s = input("enter an integer: ")
try:
    i = int(s)
```
```python
print("valid integer entered:", i)
except ValueError as err:
    print(err)
```

If the user enters “3.5”, the output will be:

```
invalid literal for int() with base 10: '3.5'
```

But if they were to enter “13”, the output will be:

```
valid integer entered: 13
```

Many books consider exception-handling to be an advanced topic and defer it as late as possible. But raising and especially handling exceptions is fundamental to the way Python works, so we make use of it from the beginning. And as we shall see, using exception handlers can make code much more readable, by separating the “exceptional” cases from the processing we are really interested in.

**Piece #6: Arithmetic Operators**

Python provides a full set of arithmetic operators, including binary operators for the four basic mathematical operations: + addition, – subtraction, * multiplication, and / division. In addition, many Python data types can be used with augmented assignment operators such as += and *=. The +, –, and * operators all behave as expected when both of their operands are integers:

```
>>> 5 + 6
11
>>> 3 – 7
-4
>>> 4 * 8
32
```

Notice that – can be used both as a unary operator (negation) and as a binary operator (subtraction), as is common in most programming languages. Where Python differs from the crowd is when it comes to division:

```
>>> 12 / 3
4.0
>>> 3 / 2
1.5
```

The division operator produces a floating-point value, not an integer; many other languages will produce an integer, truncating any fractional part. If we need an integer result, we can always convert using int() (or use the truncating division operator //, discussed later).
At first sight the preceding statements are unsurprising, particularly to those familiar with C-like languages. In such languages, augmented assignment is shorthand for assigning the results of an operation—for example, \( a += 8 \) is essentially the same as \( a = a + 8 \). However, there are two important subtleties here, one Python-specific and one to do with augmented operators in any language.

The first point to remember is that the \texttt{int} data type is immutable—that is, once assigned, an \texttt{int}’s value cannot be changed. So, what actually happens behind the scenes when an augmented assignment operator is used on an immutable object is that the operation is performed, and an object holding the result is created; and then the target object reference is re-bound to refer to the result object rather than the object it referred to before. So, in the preceding case when the statement \( a += 8 \) is encountered, Python computes \( a + 8 \), stores the result in a new \texttt{int} object, and then rebinds \( a \) to refer to this new \texttt{int}. (And if the original object \( a \) was referring to has no more object references referring to it, it will be scheduled for garbage collection.) Figure 1.3 illustrates this point.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1.3.png}
\caption{Augmented assignment of an immutable object}
\end{figure}

The second subtlety is that \texttt{a operator=b} is not quite the same as \texttt{a = a operatorb}. The augmented version looks up \( a \)’s value only once, so it is potentially faster. Also, if \( a \) is a complex expression (e.g., a list element with an index position calculation such as \texttt{items[offset + index]}), using the augmented version may be less error-prone since if the calculation needs to be changed the maintainer has to change it in only one rather than two expressions.

Python overloads (i.e., reuses for a different data type) the \texttt{+} and \texttt{+=} operators for both strings and lists, the former meaning concatenation and the latter meaning append for strings and extend (append another list) for lists:
>>> name = "John"
>>> name + "Doe"
'John Doe'
>>> name += "Doe"
>>> name
'John Doe'

Like integers, strings are immutable, so when `+=` is used a new string is created and the expression’s left-hand object reference is re-bound to it, exactly as described earlier for `int`s. Lists support the same syntax but are different behind the scenes:

>>> seeds = ["sesame", "sunflower"]
>>> seeds += ["pumpkin"]
>>> seeds
['sesame', 'sunflower', 'pumpkin']

Since lists are mutable, when `+=` is used the original list object is modified, so no rebinding of `seeds` is necessary. Figure 1.4 shows how this works.

**Figure 1.4. Augmented assignment of a mutable object**

Since Python’s syntax cleverly hides the distinction between mutable and immutable data types, why does it need both kinds at all? The reasons are mostly about performance. Immutable types are potentially a lot more efficient to implement (since they never change) than mutable types. Also, some collection data types, for example, sets, can work only with immutable types. On the other hand, mutable types can be more convenient to use. Where the distinction matters, we will discuss it—for example, in Chapter 4 when we discuss setting default arguments for custom functions, in Chapter 3 when we discuss lists, sets, and some other data types, and again in Chapter 6 when we show how to create custom data types.

The right-hand operand for the list `+=` operator must be an iterable; if it is not an exception is raised:

```python
>>> seeds += 5
Traceback (most recent call last):
...    TypeError: 'int' object is not iterable
```
The correct way to extend a list is to use an iterable object, such as a list:

```python
>>> seeds += [5]
```

```python
>>> seeds
['sesame', 'sunflower', 'pumpkin', 5]
```

And of course, the iterable object used to extend the list can itself have more than one item:

```python
>>> seeds += [9, 1, 5, "poppy"]
```

```python
>>> seeds
['sesame', 'sunflower', 'pumpkin', 5, 9, 1, 5, 'poppy']
```

Appending a plain string—for example, "durian"—rather than a list containing a string, ["durian"], leads to a logical but perhaps surprising result:

```python
>>> seeds = ['sesame', 'sunflower', 'pumpkin']
>>> seeds += "durian"
```

```python
>>> seeds
['sesame', 'sunflower', 'pumpkin', 'd', 'u', 'r', 'i', 'a', 'n']
```

The list `+=` operator extends the list by appending each item of the iterable it is provided with; and since a string is an iterable, this leads to each character in the string being appended individually. If we use the list `append()` method, the argument is always added as a single item.

### Piece #7: Input/Output

To be able to write genuinely useful programs we must be able to read input—for example, from the user at the console, and from files—and produce output, either to the console or to files. We have already made use of Python’s built-in `print()` function, although we will defer covering it further until Chapter 4. In this subsection we will concentrate on console I/O, and use shell redirection for reading and writing files.

Python provides the built-in `input()` function to accept input from the user. This function takes an optional string argument (which it prints on the console); it then waits for the user to type in a response and to finish by pressing Enter (or Return). If the user does not type any text but just presses Enter, the `input()` function returns an empty string; otherwise, it returns a string containing what the user typed, without any line terminator.

Here is our first complete “useful” program; it draws on many of the previous pieces—the only new thing it shows is the `input()` function:
print("Type integers, each followed by Enter; or just Enter to finish")

total = 0
count = 0

while True:
    line = input("integer: ")
    if line:
        try:
            number = int(line)
        except ValueError as err:
            print(err)
            continue
        total += number
        count += 1
    else:
        break

if count:
    print("count =", count, "total =", total, "mean =", total / count)

The program (in file sum1.py in the book’s examples) has just 17 executable lines. Here is what a typical run looks like:

Type integers, each followed by Enter; or just Enter to finish
integer: 12
integer: 7
integer: 1x
invalid literal for int() with base 10: '1x'
integer: 15
integer: 5
integer:
count = 4 total = 39 mean = 9.75

Book’s examples 3

Although the program is very short, it is fairly robust. If the user enters a string that cannot be converted to an integer, the problem is caught by an exception handler that prints a suitable message and switches control to the start of the loop (“continues the loop”). And the last if statement ensures that if the user doesn’t enter any numbers at all, the summary isn’t output, and division by zero is avoided.

File handling is fully covered in Chapter 7; but right now we can create files by redirecting the print() functions’ output from the console. For example:

C:\>test.py > results.txt
will cause the output of plain `print()` function calls made in the fictitious `test.py` program to be written to the file `results.txt`. This syntax works in the Windows console (usually) and in Unix consoles. For Windows, we must write `C:\Python31\python.exe test.py > results.txt` if Python 2 is the machine’s default Python version or if the console exhibits the file association bug; otherwise, assuming Python 3 is in the PATH, `python test.py > results.txt` should be sufficient, if plain `test.py > results.txt` doesn’t work. For Unices we must make the program executable (`chmod +x test.py`) and then invoke it by typing `./test.py` unless the directory it is in happens to be in the PATH, in which case invoking it by typing `test.py` is sufficient.

**Windows file association bug**

Reading data can be achieved by redirecting a file of data as input in an analogous way to redirecting output. However, if we used redirection with `sum1.py`, the program would fail. This is because the `input()` function raises an exception if it receives an EOF (end of file) character. Here is a more robust version (`sum2.py`) that can accept input from the user typing at the keyboard, or via file redirection:

```python
print("Type integers, each followed by Enter; or ^D or ^Z to finish")

total = 0
count = 0

while True:
    try:
        line = input()
        if line:
            number = int(line)
            total += number
            count += 1
        except ValueError as err:
            print(err)
            continue
        except EOFError:
            break
    if count:
        print("count =", count, "total =", total, "mean =", total / count)
```

Given the command line `sum2.py < data\sum2.dat` (where the `sum2.dat` file contains a list of numbers one per line and is in the examples’ `data` subdirectory), the output to the console is:

```
Type integers, each followed by Enter; or ^D or ^Z to finish
count = 37 total = 1839 mean = 49.7027027027
```
We have made several small changes to make the program more suitable for use both interactively and using redirection. First, we have changed the termination from being a blank line to the EOF character (Ctrl+D on Unix, Ctrl+Z, Enter on Windows). This makes the program more robust when it comes to handling input files that contain blank lines. We have stopped printing a prompt for each number since it doesn’t make sense to have one for redirected input. And we have also used a single try block with two exception handlers.

Notice that if an invalid integer is entered (either via the keyboard or due to a “bad” line of data in a redirected input file), the int() conversion will raise a ValueError exception and the flow of control will immediately switch to the relevant except block—this means that neither total nor count will be incremented when invalid data is encountered, which is exactly what we want.

We could just as easily have used two separate exception-handling try blocks instead:

```python
while True:
    try:
        line = input()
        if line:
            try:
                number = int(line)
            except ValueError as err:
                print(err)
                continue
            total += number
            count += 1
        except EOFError:
            break
```

But we preferred to group the exceptions together at the end to keep the main processing as uncluttered as possible.

**Piece #8: Creating and Calling Functions**

It is perfectly possible to write programs using the data types and control structures that we have covered in the preceding pieces. However, very often we want to do essentially the same processing repeatedly, but with a small difference, such as a different starting value. Python provides a means of encapsulating suites as functions which can be parameterized by the arguments they are passed. Here is the general syntax for creating a function:

```python
def functionName(arguments):
    suite
```

The arguments are optional and multiple arguments must be comma-separated. Every Python function has a return value; this defaults to None unless we return from the function using the syntax return value, in which case value is returned. The return value can be just one value or a tuple of values. The return value can be ignored by the caller, in which case it is simply thrown away.
Note that \texttt{def} is a statement that works in a similar way to the assignment operator. When \texttt{def} is executed a function object is created and an object reference with the specified name is created and set to refer to the function object. Since functions are objects, they can be stored in collection data types and passed as arguments to other functions, as we will see in later chapters.

One frequent need when writing interactive console applications is to obtain an integer from the user. Here is a function that does just that:

```python
def get_int(msg):
    while True:
        try:
            i = int(input(msg))
            return i
        except ValueError as err:
            print(err)
```

This function takes one argument, \texttt{msg}. Inside the \texttt{while} loop the user is prompted to enter an integer. If they enter something invalid a \texttt{ValueError} exception will be raised, the error message will be printed, and the loop will repeat. Once a valid integer is entered, it is returned to the caller. Here is how we would call it:

```python
age = get_int("enter your age: ")
```

In this example the single argument is mandatory because we have provided no default value. In fact, Python supports a very sophisticated and flexible syntax for function parameters that supports default argument values and positional and keyword arguments. All of the syntax is covered in \textbf{Chapter 4}.

Although creating our own functions can be very satisfying, in many cases it is not necessary. This is because Python has a lot of functions built in, and a great many more functions in the modules in its standard library, so what we want may well already be available.

A Python module is just a \texttt{.py} file that contains Python code, such as custom function and class (custom data type) definitions, and sometimes variables. To access the functionality in a module we must import it. For example:

```python
import sys
```

To import a module we use the \texttt{import} statement followed by the name of the \texttt{.py} file, but omitting the extension. Once a module has been imported, we can access any functions, classes, or variables that it contains. For example:
The sys module, some other built-in modules, and modules implemented in C don’t necessarily have corresponding .py files—but they are used in just the same way as those that do.

print(sys.argv)

The sys module provides the argv variable—a list whose first item is the name under which the program was invoked and whose second and subsequent items are the program’s command-line arguments. The two previously quoted lines constitute the entire echoargs.py program. If the program is invoked with the command line echoargs.py -v, it will print ['echoargs.py', '-v'] on the console. (On Unix the first entry may be './echoargs.py'.)

In general, the syntax for using a function from a module is

moduleName.functionName(arguments).

It makes use of the dot (“access attribute”) operator we introduced in Piece #3. The standard library contains lots of modules, and we will make use of many of them throughout the book. The standard modules all have lowercase names, so some programmers use title-case names (e.g., MyModule) for their own modules to keep them distinct.

Dot (. ) operator 21

Let us look at just one example, the random module (in the standard library’s random.py file), which provides many useful functions:

import random
x = random.randint(1, 6)
y = random.choice(['apple', 'banana', 'cherry', 'durian'])

After these statements have been executed, x will contain an integer between 1 and 6 inclusive, and y will contain one of the strings from the list passed to the random.choice() function.

It is conventional to put all the import statements at the beginning of .py files, after the shebang line, and after the module’s documentation. (Documenting modules is covered in Chapter 5.) We recommend importing standard library modules first, then third-party library modules, and finally your own modules.
Related Content

- **Python Language Structure**
  From: *Game Programming with Python, Lua, and Ruby*

- **Why learn Python 3?**
  From: *The Quick Python Book, Second Edition*

- **An overview of Python**
  From: *IronPython in Action*