

1.5 Basic Elements of a Program



Every programming language has a **syntax**: rules that describe the structure of valid combinations of characters and words in a program.

When one first begins writing in a programming language, it is common to generate **syntax errors**, improper combinations of characters and words. Every programming language also has a **semantics**: a meaning associated with a syntactically valid program. A program's semantics describe what a program does.

In Python and in other programming languages, programs are composed of a sequence of smaller elements called **statements**. Statements do something, like perform a calculation or store a value in memory. For instance, `x = 3*5` is a statement that computes a product and stores the result under the variable name `x`. Many statements contain **expressions**, each of which produces a value. For instance, `3*5` in the previous statement is an expression that produces the value 15.

An expression contains smaller elements called **operands** and **operators**. Common operands include **identifiers**—names like variables, functions, and modules that refer to objects—and **literals**—notations for constant values of a built-in type. For instance, in the previous expression `x` is a variable identifier and 3 and 5 are literals that evaluate to objects of the built-in `integer` class. The `*` character in the previous expression is the multiplication operator. Python includes operators for arithmetic (e.g., `+`), assignment (e.g., `=`), comparison (e.g., `>`), logic (e.g., `or`), identification (e.g., `is`), membership (e.g., `in`), and other operations.

Example 1.1

Create a Python program that computes the following arithmetic expressions:

$$x = 4069 \cdot 0.002, \quad y = 100/1.5, \text{ and } \quad z = (-3)^2 + 15 - 3.01 \cdot 10.$$

Multiply these together (`x*y*z`) and print the product, along with `x`, `y`, and `z` to the console.

Consider the following program:

```
x = 4096*0.002           # float multiplication
y = 100/1.5              # float division
z = (-3)**2 + 15 - 3.01*10 # exponent operator **
print(x,y,z)
print(x*y*z)
```

The console should print

```
8.192 66.66666666666667 -6.099999999999999
-3331.413333333333
```

Note that although we have multiplied and divided integer literals (4096 and 100) by floating-point literals (0.002 and 1.5), Python has automatically assumed we would like floating-point multiplication and division.

We used the exponent operator `**`, which may have been unfamiliar. If you tried the more common character `^` for the exponent, you received the error

```
| TypeError: unsupported operand type(s) for ^: 'int' and 'float'
```

In Python, the `^` is the bitwise logical XOR operator.

1.5.1 Classes, Objects, and Methods

Everything that is expressed in a Python statement is an **object**, and every object is an **instance** of a **class**. For instance, `7` is a literal that evaluates to an object that is an instance of the `integer` class. Similarly, `"foo"` is a literal that evaluates to an object that is an instance of the `string` class. A class can be thought of as a definition of the kind of objects that belong to it, how they are structured, and the kinds of things that can be done with it.

Python includes built-in classes such as the numeric `integer`, `floating-point` number, and `complex` number. It is common to refer to a class, especially a built-in class, as a **type**.

Classes are more than types of data, however. Classes can include one or more **method**, which is a kind of function that operates on inputs called **arguments** and returns outputs. Something special about methods is that they can operate on instances of the class. For example,

```
| 3.5.as_integer_ratio()
```

The literal `3.5` yields an instance of the `float` class, which has method `as_integer_ratio()`. Placing the `.` character before the method name is the syntax to apply the object's `as_integer_ratio()` method. This method returns a `tuple` object with the form `(<numerator>, <denominator>)`, where `<numerator>` and `<denominator>` denote the numerator and denominator of the integer ratio corresponding to the floating-point number. The expression yields the output

```
| (7, 2)
```

which signifies the fraction $7/2$.

We can and often do create our own classes with their own methods. We will return to this topic in a later chapter.

Example 1.2

Create a Python program that starts with the three word strings "veni", "vedi", "vici" and concatenates and prints them with the following caveats:

- Between each word string, insert a comma and a space.
- Capitalize each word string using the `capitalize()` method.

Consider the following program:

```
w1 = "veni"
w2 = "vedi"
w3 = "vici"
print(w1.capitalize() + ", " +
      w2.capitalize() + ", " +
      w3.capitalize()
)
```

The console should print

```
| Veni, Vedi, Vici
```

Note that we have used linebreaks to improve code readability. Python syntax allows expressions enclosed in parentheses to be broken after operators.

1.5.2 Basic Built-In Types

Python has several built-in types (classes) that provide a foundation from which many of our programs can be written. We have seen some examples of these types already, and in this section they will be described in greater detail.

1.5.2.1 Boolean The simple `bool` (i.e., Boolean) type can have one of two values, `True` or `False`. This type is used extensively for logical reasoning in programs, and will be especially important for branching (see section 1.10). Expressions containing the **Boolean operators** `not`, `and`, and `or` evaluate to Boolean values. For instance,

```
not True      # => False
not False     # => True
True and False # => False
True or False  # => True
```

Similarly, expressions with the **comparison operators** `==` (equality), `!=` (inequality), `<` (less than), `>` (greater than), `<=` (less than or equal), `>=` (greater than or equal), `is` (identity), `is not` (nonidentity), and `in` (membership). evaluate to Boolean values. A **truth table** for the Boolean operators and some comparison operators is given in table 1.1.

Table 1.1. Boolean and comparison operators on Boolean and integer inputs x and y

x	y	<code>bool(x)</code>	<code>not x</code>	<code>x and y</code>	<code>x or y</code>	<code>x==y</code>	<code>x!=y</code>	<code>x<y</code>	<code>x<=y</code>	<code>x>=y</code>	<code>x>y</code>
False	False	False	True	False	False	True	False	False	True	True	False
False	True	False	True	False	True	False	True	True	True	False	False
True	False	True	False	False	True	False	True	False	False	True	True
True	True	True	False	True	True	True	False	False	True	True	False
0	0	False	True	0	0	True	False	False	True	True	False
0	1	False	True	0	1	False	True	True	True	False	False
1	0	True	False	0	1	False	True	False	False	True	True
1	1	True	False	1	1	True	False	False	True	True	False

Note that non-Boolean inputs can be given to the Boolean operators. Non-Boolean objects can be given Boolean values with the `bool()` function, included in the table. For instance, `bool(0)` and `bool(0.0)` evaluate to **False**; conversely, `bool(1)` and `bool(1.0)` evaluate to **True**. In fact, for all numeric types (i.e., `int`, `float`, and `complex`), every value evaluates to **True** except those equivalent to 0.

1.5.2.2 Integer The `int` (i.e., integer) type can be used to represent the mathematical integers, positive and negative (and 0). As we have already seen, several built-in operators can be applied to integer inputs, including `+` (summation), `-` (difference), `*` (product), and `/` (quotient). More operators will be introduced in later chapters.

The built-in `int()` function returns an integer representation of its input, which can be either a number, a string, or empty, in which case `int()` returns 0. Although it does not round in the most elegant manner, `int()` can be used to convert a floating-point number to an integer, as in

```
int(3.2)      # => 3
int(3.9)      # => 3
int(-3.9)     # => -3
```

We see that `int()` rounds toward zero.

1.5.2.3 Floating-Point Number Like scientific notation, **floating-point numbers** represent potentially very large or very small numbers in a compact form. This form has three parts: a **sign** s , a **significand** x , and an **exponent** n . These combine as

$$s \times x \times 2^n.$$

Floating-point numbers can be represented with the Python `float` type and are often used to represent decimal numbers, such as 1.4 and -0.33 . The built-in `float()` function returns a `float` from a number or string argument. For instance,

```
float(5)      # => 5.0
float("5")    # => 5.0
```

Floating-point numbers can be entered with scientific notation via the letter `e`, as in the following examples:

```
| 291e-6      # => 0.000291
| 1e3         # => 1000.0
```

1.5.2.4 Complex Number Complex numbers, which have a real part a and imaginary part b , represented mathematically as

$$a + jb,$$

where j is the imaginary number $\sqrt{-1}$, can be represented in Python with the `complex` type. For numbers `a` and `b`, we can construct a `complex` type with `complex(a, b)`. For instance,

```
| complex(1, 2)  # => (1+2j)
```

A `complex` object has attributes `real` and `imag` that return the real and imaginary parts, respectively. For instance,

```
| s = complex(3, -5)
| s.real      # => 3
| s.imag      # => -5
```

1.5.2.5 String Strings are series of characters and have built-in Python type `str`. String literals can be written with either single quotes (e.g., `'foo'`) or double quotes (e.g., `"bar"`). Within one variety, the other is treated as a regular quote, as in `"A 'friend' wants to know"` and `'I am "big boned"'`. It is generally recommended to use just one variety or the other for string literals in a given project (mixing the two is seen as bad form).

The `str()` function returns a string representation of the object it is given as input. For instance, `str(4)` returns `"4"` and `str(True)` returns `"True"`. This is especially helpful when joining strings as in the following example:

```
| x = 3.14159
| print("x = " + str(x) + " m")
```

A convenient way to construct nice strings is the **formatted string (f-string)** literals. A simple f-string that has the same value as what is printed in the example above is `f"x = {x} m"`. Executable expressions are inserted in braces `{}` within the f-string. Note that the `str()` function is automatically called, which makes for a nicer syntax.

A **format specifier** can also be applied to expressions in an f-string. These have the general form

```
| : [[fill]align] [sign] [z] [#] [0] [width] [group] [.prec] [type]
```

Each of these terms is described in table 1.2 and table 1.3. In the example above, we could format the printing of `x` in fixed-point format (`f`) with a precision (`.prec`) of 3 decimal places with

```
| print(f"x = {x:.3f} m")      # => x = 3.142
```

In the following example, we use scientific notation (`e`) with precision (`.prec`) of 4:

```
| x = 0.00123
| print(f"x = {x:.4e} m")      # => x = 1.2300e-03 m
```

Note that the number of **significant digits** is 1 greater than the precision in this format. In the following example, we use binary (`b`) with 0-padding:

```
| x = 3
| print(f"x = {x:04b} (in binary)")      # => x = 0011 (in binary)
```

Table 1.2. Format specifier terms.

Term	Values (if any)	Default	Effect
:			Separates the format specifier from the expression
fill	Any character	space	Character to pad with when value doesn't use the entire field width
align	< (left) > (right) ^ (center) =	<	How to justify when value doesn't occupy the entire field width
sign	+ (explicit +) - (no plus) space (space but no plus)	-	How a sign appears for numeric values
z	z		Coerces -0.0 to 0.0
#	#		Use the alternate output form for numeric values
0	0		Pad on the left with zeros instead of spaces
width	Positive integers		Minimum width of (number of characters in) the field
group	, _		Grouping character (thousands separator) for numeric output
.prec	Nonnegative integers	varies with <code>type</code>	Digits after the decimal point for floating-points, maximum width for strings
type	See table 1.3	s (strings) or d (numbers)	Specifies the presentation type, which is the type of conversion performed on the corresponding argument

Table 1.3. Format specifier types.

Input Class	Format Type	Meaning
String	s	String
String	None	String (same as s)
Integer	b	Binary

Input Class	Format Type	Meaning
Integer	c	Character
Integer	d	Decimal integer
Integer	o	Octal
Integer	x or X	Hex (lowercase or uppercase)
Integer	n	Local decimal number (similar to d)
Integer	None	Same as d
Floating-point	e or E	Scientific notation (lowercase or uppercase)
Floating-point	f or F	Fixed-point notation (lowercase or uppercase)
Floating-point	g or G	General format (lowercase or uppercase)
Floating-point	n	Local general format
Floating-point	%	Percentage
Floating-point	None	Same as g

The `str` class has several methods. We have already seen the `capitalize()` method applied in section 2.3.2. Table 1.4 describes several frequently used string methods.

Table 1.4. Some particularly useful string methods.

Method	Description
<code>capitalize()</code>	Convert the first character to uppercase
<code>count()</code>	Return the count of the specified value occurrences
<code>endswith()</code>	If the string ends with the specified value, return True
<code>find()</code>	Return the position where the specified value is found
<code>index()</code>	Return the position where the specified value is found
<code>isalpha()</code>	If all characters are alphabetic, return True
<code>isdecimal()</code>	If all characters are decimals, return True
<code>isdigit()</code>	If all characters are digits, return True
<code>isnumeric()</code>	If all characters are numeric, return True
<code>join()</code>	Convert iterable elements into a single string
<code>lower()</code>	Convert the string to lowercase
<code>replace()</code>	Return a string with the specified value replaced
<code>rindex()</code>	Return the last position where the specified value is found
<code>rsplit()</code>	Split at the specified separator, return a list
<code>split()</code>	Split at the specified separator, return a list
<code>splitlines()</code>	Split at line breaks, return a list
<code>startswith()</code>	If the string starts with the specified value, return True
<code>strip()</code>	Return a trimmed version of the string

1.5.3 Iterable Objects and Dictionaries

In Python, an **iterable object** is one that contains a collection of **elements** and defines, for each element, which element is next. In the following sections, we will consider some built-in iterable classes (types).

Box 1.1 Further Reading

- Python Community (2024a; § The Python Tutorial: 9 Classes), on classes, objects, and methods
- Python Community (2024a; § Python Standard Library: Built-in Types), on the basic built-in types

1.6 Lists



The **list** class defines an ordered set of elements. These elements can be of any class, and do not need to match within a list. Lists can be nested to create a list of lists. The basic syntax for creating a list of elements *ex* is `[e1, e2, ..., en]`. Consider the following list assignments:

```
int_list = [3, 9, 3, -4, 0]           # Duplication allowed
str_list = ["foo", "bar", "baz"]
com_list = [int_list, str_list]      # List of lists
mix_list = [8.41, "foo", [7]]       # Mixing element types
```

1.6.1 Accessing List Elements

Because the elements of a list have an order, they can be referred to via an **index**, a mapping of integers to elements. In Python, the first element in the list has index 0 and subsequent elements have indices of increasing values, 1, 2, 3, and so on. The syntax for accessing the element with index *i* of a list *l* is `l[i]`. For instance, elements from the previously defined lists can be accessed as follows:

```
int_list[0]           # => 3
int_list[3]           # => -4
str_list[2]           # => "baz"
mix_list[2]           # => [7]
```

Negative indices are used to access elements from the end of a list. For instance, for `int_list` above,

```
int_list[-1]          # => 0
int_list[-2]          # => -4
```

This is particularly useful when we want to access the last element of a list, which we see has index `-1`.

A selection of elements from a list can be accessed via **slicing**, which has the syntax `l[start:stop]` or `l[start:stop:step]`. For instance,