Literals

To take something literally is to take it at "face value." The same is true of *literals* in programming. A **literal** is a sequence of one or more characters that stands for itself, such as the literal 12. We look at numeric literals in Python next.

Part I - Numeric Literals

A **numeric literal** is a literal containing only the digits 0–9, an optional sign character (1 or 2), and a possible decimal point. (The letter *e* is also used in exponential notation, shown in the next subsection). If a numeric literal contains a decimal point, then it denotes a **floating-point value**, or "**float**" (e.g., 10.24); otherwise, it denotes an **integer value** (e.g., 10). *Commas are never used in numeric literals*. The chart below gives additional examples of numeric literals in Python.

Numeric Literals		
integer values	floatirg-point values	incorrect
5	5. 5.0 5.125 0.0005 5000.125	5,000.125
2500	2500. 2500.0 2500.125	2,500 2,500.125
+2500	+2500. +2500.0 +2500.125	+2,500 +2,500.125
-2500	-2500. -2500.0 -2500.125	-2,500 -2,500.125

Since numeric literals without a provided sign character denote positive values, an explicit positive sign character is rarely used. Next we look at how numeric values are represented in a computer system.

Your Turn

From the Python Shell, enter the following:

```
… 1024
???

… 21024
???

… .1024
???

… 1,024
???

… 0.1024
???

… 1,024.46
???
```

Limits of Range in Floating-Point Representation

There is no limit to the size of an integer that can be represented in Python. Floating-point values, however, have both a limited *range* and a limited *precision*. Python uses a double-precision standard format (IEEE 754) providing a range of 102308 to 10308 with 16 to 17 digits of precision. To denote such a range of values, floating-points can be represented in scientific notation,

`9.0045602e15` (9.0045602 3 105, 8 digits of precision)

`1.006249505236801e8` (1.006249505236801 3 108, 16 digits of precision)

`4.239e216` (4.239 3 10216, 4 digits of precision)

It is important to understand the limitations of floating-point representation. For example, the multiplication of two values may result in **arithmetic overflow**, a condition that occurs when a calculated result is too large in magnitude (size) to be represented,

```
>>> 1.5e200 * 2.0e210
>>> inf
```

This results in the special value `inf` ("infinity") rather than the arithmetically correct result `3.0e410`, indicating that arithmetic overflow has occurred. Similarly, the division of two numbers may result in **arithmetic underflow**, a condition that occurs when a calculated result is too small in magnitude to be represented,

```
>>>1.0e2300 / 1.0e100 0.0
```

This results in `0.0` rather than the arithmetically correct result `1.0e2400`, indicating that arithmetic underflow has occurred. We next look at possible effects resulting from the limited precision in floating-point representation.

Your Turn

From the Python Shell, enter the following and observe the results.

```
>>>1.2e200 * 2.4e100        >>>1.2e200 / 2.4e100
???                         ???
>>>1.2e200 * 2.4e200        >>>1.2e2200 / 2.4e200
```

??? ???

Limits of Precision in Floating-Point Representation

Arithmetic overflow and arithmetic underflow are relatively easily detected. The loss of precision that can result in a calculated result, however, is a much more subtle issue. For example, 1/3 is equal to the infinitely repeating decimal .33333333 . . ., which also has repeating digits in base two,.010101010. . . . Since any floating-point representation necessarily contains only a finite number

of digits, what is stored for many floating-point values is only an *approximation* of the true value, as can be demonstrated in Python,

```
>>>1/3
.3333333333333333
```

Here, the repeating decimal ends after the 16th digit. Consider, therefore, the following,

```
>>>3 * (1/3) 1.0
```

Given the value of 1/3 above, we would expect the result to be .9999999999999999, so what is happening here? The answer is that Python displays a *rounded* result to keep the number of digits displayed manageable. However, the representation of 1/3 as .3333333333333333 remains the same, as demonstrated by the following,

```
>>>1/3 1 1/3 1 1/3 1 1/3 1 1/3 1 1/3
1.9999999999999998
```

In this case we get a result that reflects the representation of 1/3 as an approximation, since the last digit is 8, and not 9. However, if we use multiplication instead, we again get the rounded value displayed,

```
>>>6 * (1/3)
2.0
```

The bottom line, therefore, is that no matter how Python chooses to display calculated results, the value stored is limited in both the range of numbers that can be represented and the degree of precision. For most everyday applications, this slight loss in accuracy is of no practical concern.

However, in scientific computing and other applications in which precise calculations are required, this is something that the programmer must be keenly aware of.

Your Turn

From the Python Shell, enter the following and observe the results.

```
>>> 1/10
???
>>> 1/10 + 1/10 + 1/10
???
>>> 10 * (1/10)
???
>>> 6 * (1/10)
???
>>> 6 * 1/10
???
```

Since any floating-point representation contains only a finite number of digits, what is stored for many floating-point values is only an *approximation* of the true value.

Built-in `format` Function

Because floating-point values may contain an arbitrary number of decimal places, the built-in **`format`** function can be used to produce a numeric string version of the value containing a specific number of decimal places,

```
>>> 12/5
2.4
>>> 5/7
0.7142857142857143

>>> format(12/5, '.2f')
'2.40'
>>> format(5/7, '.2f')
'0.71'
```

In these examples, *format specifier* `'.2f'` rounds the result to two decimal places of accuracy in the string produced. For very large (or very small) values `'e'` can be used as a format specifier,

... `format(2 ** 100, '.6e') '1.267651e130'`

In this case, the value is formatted in scientific notation, with six decimal places of precision. Formatted numeric string values are useful when displaying results in which only a certain number of decimal places need to be displayed,

without use of format specifier

```
>>> tax = 0.08
>>> print('Your cost: $', (1 + tax) * 12.99)
Your cost: $ 14.029200000000001
```

with use of specifier

```
>>> print('Your cost: $', format((1 + tax) * 12.99, '.2f'))
Your cost: $ 14.03
```

Finally, a comma in the format specifier adds comma separators to the result,

```
>>> format(13402.25, ',.2f') 13,402.24
```

We will next see the use of format specifiers for formatting string values as well.

Your Turn

From the Python Shell, enter the following and observe the results.

```
>>> format(11/12, '.2f')      >>> format(11/12, '.2e')
???                           ???
>>> format(11/12, '.3f')      >>> format(11/12, '.3e')
???                           ???
```

Part II - String Literals

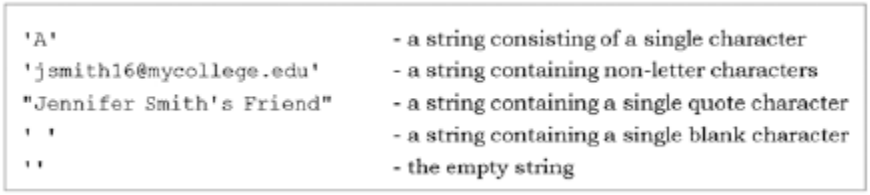
Numerical values are not the only literal values in programming. **String literals**, or "**strings**," rep- resent a sequence of characters,

```
'Hello' 'Smith, John' "Baltimore, Maryland 21210"
```

In Python, string literals may be *delimited* (surrounded) by a matching pair of either single (') or double (") quotes. Strings must be contained all on one line (except when delimited by triple quotes.) We have already seen the use of strings in Chapter 1 for displaying screen output,

```
>>>print('Welcome to Python!') Welcome to Python!
```

Additional examples of string literals are given below:



As shown in the figure, a string may contain zero or more characters, including letters, digits, special characters, and blanks. A string consisting of only a pair of matching quotes (with nothing in between) is called the **empty string**, which is different from a string containing only blank characters. Both blank strings and the empty string have their uses, as we will see. Strings may also contain quote characters as long as different quotes are used to delimit the string,

```
"Jennifer Smith's Friend"
```

If this string were delimited with single quotes, the apostrophe (single quote) would be considered the matching closing quote of the opening quote, leaving the last final quote unmatched,

```
'Jennifer Smith's Friend' ... matching quote?
```

Thus, Python allows the use of more than one type of quote for such situations. (The convention used in the text will be to use single quotes for delimiting strings, and only use double quotes when needed.)

Your Turn

From the Python Shell, enter the following and observe the results.

```
>>>print('Hello')    >>>print('Hello”)    >>>print(“Hello”)
>>>print('Friend')   >>>print(“Friend’)   >>>print(“Friend”)
```

A **string literal**, or **string**, is a sequence of characters denoted by a pair of matching single or double (and sometimes triple) quotes in Python.

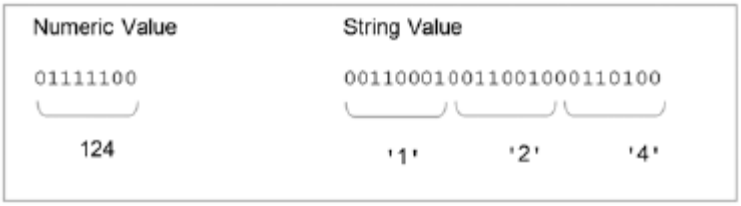
The Representation of Character Values

There needs to be a way to encode (represent) characters within a computer. Although various encoding schemes have been developed, the **Unicode** encoding scheme is intended to be a universal encoding scheme. Unicode is actually a collection of different encoding schemes utilizing between 8 and 32 bits for each character. The default encoding in Python uses **UTF-8**, an 8-bit encoding compatible with ASCII, an older, still widely used encoding scheme.

Currently, there are over 100,000 Unicode-defined characters for many of the languages around the world. Unicode is capable of defining more than 4 billion characters. Thus, all the world’s languages, both past and present, can potentially be encoded within Unicode. A partial listing of the ASCII-compatible UTF-8 encoding scheme is given below:

Space	00100000	32	A	01000001	65
!	00100001	33	B	01000010	66
"	00100010	34	C	01000011	67
#	00100011	35	.		
.			.		
.			Z	01011010	90
0	00110000	48	a	01100001	97
1	00110001	49	b	01100010	98
2	00110010	50	c	01100011	99
.			.		
.			.		
9	00111001	57	z	01111010	122

UTF-8 encodes characters that have an ordering with sequential numerical values. For example, `'A'` is encoded as `01000001` (65), `'B'` is encoded as `01000010` (66), and so on. This is true for character digits as well, `'0'` is encoded as `00110000` (48) and `'1'` is encoded as `00110001` (49). This underscores the difference between a numeric representation (that can be used in arithmetic calculations) vs. a number represented as a string of digit characters (that cannot), as demonstrated in Figure 2-5.



Python has means for converting between a character and its encoding. The `ord` function gives the UTF-8 (ASCII) encoding of a given character. For example, `ord('A')` is 65. The `chr` function gives the character for a given encoding value, thus `chr(65)` is `'A'`. While in general there is no need to know the specific encoding of a given character, there are times when such knowledge can be useful.

Your Turn

From the Python Shell, enter the following and observe the results.

```
>>>ord('14')        >>>chr('35')        >>>chr('68')
???                 ???                 ???

>>>ord('7')         >>>chr('83')        >>>chr('128')
???                 ???                 ???
```

Part III - Control Characters

Control characters are special characters that are not displayed on the screen. Rather, they *control* the display of output (among other things). Control characters do not have a corresponding keyboard character. Therefore, they are represented by a combination of characters called an *escape sequence*.

An **escape sequence** begins with an **escape character** that causes the sequence of characters following it to "escape" their normal meaning. The backslash (\) serves as the escape character in Python. For example, the escape sequence `'\n'`, represents the *newline control character*, used to begin a new screen line. An example of its use is given below,

```
print('Hello\nJennifer Smith')
```

which is displayed as follows,

```
Hello Jennifer Smith
```

Your Turn

From the Python Shell, enter the following and observe the results.

```
>>> print('Hello World')
???

>>> print('Hello\nWorld')
???

>>> print('Hello World\n')
???

>>> print('Hello\n\nWorld')
???

>>> print('Hello World\n\n')
???

>>> print(1, '\n', 2, '\n', 3)
???

>>> print('\nHello World')
???

>>> print('\n',1,'\n',2,'\n', 3)
???
```

Part IV - String Formatting

We saw above the use of built-in function `format` for controlling how numerical values are displayed. We now look at how the `format` function can be used to control how strings are displayed. As given above, the `format` function has the form,

`format(`*`value, format_specifier`*`)`

where *value* is the value to be displayed, and *format_specifier* can contain a combination of formatting options. For example, to produce the string `'Hello'` left-justified in a field width of 20 characters would be done as follows,

```
format('Hello', '<20') → 'Hello '
```

To right-justify the string, the following would be used,

```
format('Hello', '>20') → '  Hello'
```

Formatted strings are left-justified by default. To center the string the `'^'` character is used: `format('Hello', '^20')`. Another use of the `format` function is to create strings of blank characters, which is sometimes useful,

```
format(' ', '30') → '      '
```

Finally blanks, by default, are the *fill character* for formatted strings. However, a specific fill character can be specified as shown below,

```
>>> print('Hello World', format('.', '.<30'), 'Have a Nice Day!')
Hello World ............................. Have a Nice Day!
```

Your Turn

From the Python Shell, enter the following and observe the results.

```
>>> print(format('Hello World', '^40'))
???


> > > print(format ( '-', '-<20'), 'Hello World',
    format ( '-', '-<20')
???
```

Part V - Implicit and Explicit Line Joining

Sometimes a program line may be too long to fit in the Python-recommended maximum length of 79 characters. There are two ways in Python to do deal with such situations—implicit and explicit line joining. We discuss this next.

Implicit Line Joining

There are certain delimiting characters that allow a *logical* program line to span more than one physical line. This includes matching parentheses, square brackets, curly braces, and triple quotes. For example, the following two program lines are treated as one logical line,

```
print('Name:', student_name, 'Address:', student_address, 'Number
of Credits:', total_credits, 'GPA:', current_gpa)
```

Matching quotes (except for triple quotes, covered later) must be on the same physical line. For example, the following will generate an error,

```
print('This program will calculate a restaurant tab for a couple
with a gift certificate, and a restaurant tax of 3%')
```

We will use this aspect of Python throughout the course.

Explicit Line Joining

In addition to implicit line joining, program lines may be explicitly joined by use of the backslash (\) character. Program lines that end with a backslash that are not part of a literal string (that is, within quotes) continue on the following line,

```
numsecs_1900_dob = ((year_birth - 1900) * avg_numsecs_year) + \
((month_birth - 1) * avg_numsecs_month) + \ (day_birth *
numsecs_day)
```

Part VI - Let's Apply It—"Hello World Unicode Encoding"

It is a long tradition in computer science to demonstrate a program that simply displays "Hello World!" as an example of the simplest program possible in a particular programming language. In Python, the complete Hello World program is comprised of one program line,

```
print('Hello World!')
```

We take a twist on this tradition and give a Python program that displays the Unicode encoding for each of the characters in the string “Hello World!” instead. This program utilizes the following programming features:

➤ string literals ➤ `print` ➤ `ord` function

The program and program execution are given below:

Program Execution ...

```
The Unicode encoding for 'Hello World!' is:
72 101 108 108 111 32 87 111 114 108 100 33
```

```
1 # This program displays the Unicode encoding for 'Hello World!'
2
3 # Program greeting
4 print("The Unicode encoding for 'Hello World!' is:")
5
6 # output results
7 print(ord('H'), ord('e'), ord('l'), ord('l'), ord('o'), ord(' '),
8       ord('W'), ord('o'), ord('r'), ord('l'), ord('d'), ord('!'))
```

The statements on **lines 1**, **3**, and **6** are comment statements. They are ignored during program execution, used to provide information to those reading the program. The print function on **line 4** displays the message ‘Hello World!’. Double quotes are used to delimit the corresponding string, since the single quotes within it are to be taken literally. The use of `print` on **line 7** prints out the Unicode encoding, one-by-one, for each of the characters in the “Hello World!” string. Note from the program execution that there is a Unicode encoding for the blank character (32), as well as the exclamation mark (33).

Concepts and Procedures

1. Indicate which of the following are valid numeric literals in Python.

(a) `1024` **(b)** `1,024` **(c)** `1024.0` **(d)** `0.25` **(e)** `.45` **(f)** `0.25110`

2. Indicate which of the following exceed the range and/or precision of floating-point values that can be represented in Python.

(a) `1.89345348392e1301` **(c)** `2.0424e2320`

(b) `1.62123432632322e1300` **(d)** `1.32323243534232789645 2e2140`

3. Which of the following would result in either overflow or underflow for the floating-point representation scheme mentioned in the chapter.

(a) `6.25e1240 * 1.24e110` **(c)** `6.25e1240 / 1.24e110`

(b) `2.24e1240 * 1.45e1300` **(d)** `2.24e2240 / 1.45e1300`

4. Exactly what is output by `print(format(24.893952, '.3f'))`

(a) `24.894` **(b)** `24.893` **(c)** `2.48e1`

5. Which of the following are valid string literals in Python.

(a) `"Hello"` **(b)** `'hello'` **(c)** `"Hello'` **(d)** `'Hello there'` **(e)** `''`

6. Which of the following results of the ord and chr functions are correct?

(a) `ord('1')→ 49` **(b)** `chr(68)→ 'd'` **(c)** `chr(99)→ 'c'`

7. How many lines of screen output is displayed by the following,

```
print('apple\nbanana\ncherry\npeach')
```

Problem Solving

1. Give the following values in the exponential notation of Python, such that there is only one significant digit to the left of the decimal point.

(a) `4580.5034` **(b)** `0.00000046004` **(c)** `5000402.000000000006`

2. Which of the floating-point values in question 1 would exceed the representation of the precision of floating points typically supported in Python?

3. Regarding the built-in `format` function in Python,

(a) Use the `format` function to display the floating-point value in a variable named `result` with three decimal digits of precision.

(b) Give a modified version of the format function in (a) so that commas are included in the displayed results.

4. Give the string of binary digits that represents, in ASCII code,

(a) The string `'Hi!'`

(b) The literal string `'I am 24'`

6. Give a call to `print` that is provided one string that displays the following address on three separate lines.

```
John Doe

123 Main Street

Anytown, Maryland 21009
```

7. Use the `print` function in Python to output `It's raining today.`

8. Regarding variable assignment,

(a) What is the value of variables `num1` and `num2` after the following instructions are executed?

```
num 5 0

k 5 5

num1 5 num 1 k * 2

num2 5 num 1 k * 2
```

(b) Are the values `id(num1)` and `id(num2)` equal after the last statement is executed?

9. Regarding the `input` function in Python,

(a) Give an instruction that prompts the user for their last name and stores it in a variable named `last_name.`

(b) Give an instruction that prompts the user for their age and stores it as an integer value named `age`.

(c) Give an instruction that prompts the user for their temperature and stores it as a float named

```
current_temperature.
```

10. Regarding keywords and other predefined identifiers in Python, give the result for each of the following,

(a) `'int' in dir( builtins )`

(b) `'import' in dir( builtins )`