2.5 Strings

Computers were originally designed to "compute" numeric values, but at a very early stage scientists realized that they were just as useful for manipulating textual information. The basic data type for working with text is called a *string*. Python has a particularly strong set of operations for working with strings. Here we will introduce the basic techniques for strings, which will allow us to use text as well as numbers in our programs. In a later we will give a more thorough coverage of strings and their capabilities.

There are three ways you can get strings into a program: they can be literal strings in the program code, you can input them from the user or a file, or you can compute them from simpler strings, even from individual letters. String literals are sequences of characters inside quotation marks. You can use either single quotes or double quotes, as long as you end the string with the same kind of quote that you used to start it. For example "Four score and seven years ago" and 'Barack Obama' are two examples of string literals. Note that "example" and 'example' are exactly the same string. The string "", which contains no characters at all (in other words, this string has length 0) is called the empty string. Empty strings serve a role similar to 0. We often will build up strings by starting with the empty string and adding letters. Note that in Python there is no separate type for individual characters, such as the letter 'a'. Characters in Python are just strings of length 1.

You can work with strings just as you work with numbers. Variables can be assigned string values, as in

```
name = "Hieronymous Bosch"
```

Strings can be given values through user input, as in

```
name = input("Enter a name: ")
```

The **input**() function doesn't try to evaluate what the user types, it just returns it as a string. Finally, you can build up a string value within a program, as in

```
name = "Marvin " + "Krislov"
```

There are many operators and functions for working with strings. One of the most common is the length function **len**(s), which returns the number of letters in string s. The + operator when used with strings gives the concatenation. or pushing together, of the operands. For example, "multi" + "media" is the new string "multimedia". Note that you can add two numbers with + and you can concatenate two strings with +, but it doesn't make sense to use + between a string and a number.

Sometimes we want to grab the individual letters that make up a string. The expression s[n] returns the letter in the *nth* position of string s. For technical reasons Python, and most other computer languages, start indexing strings at position 0. Thus, if test is the string "abcde", then test[0] is the letter "a", test[1] is "b", and so forth. String x is indexed from position 0 to position **len**(x)−1.

A *slice* of string `s` is a substring made up of contiguous letters in `s`. `s[i:j]` is the slice of string `s` consisting of letters from position `i` to position `j` (including `i` and not including `j`). Again, we start indexing positions with 0. For example, if `word` is the string `"together"`, then `word[2:5]` is the string `"get"`.

The following table summarizes the most commonly used string operations. In this table `x`, `s`, and `t` are strings, `n`, `i`, `j`, and `k` are integers.

Symbol	Meaning	Example	Result
str(n)	Returns the string version of n.	str(23)	"23"
len(s)	Length of string s.	len("Katrina")	7
s+t	Concatenation of strings s and t.	"multi" + "media"	"multimedia"
s*n or n*s	String s is repeated n times.	"bob"*3	"bobbobbob"
s[n]	The nth letter of string s. Indexing starts at 0.	x ="abcde" x[3]	 "d"
s[i:j]	The slice of string s, consisting of the letters starting at position j (indexing starts with 0) and extending up to but not including position j.	type = "deciduous" type[4:7]	 "duo"
s[i:]	This is the slice of string s starting at position i and going to the end of s.	type = "deciduous" type[4:]	 "duous"
s[:j]	This is the slice of string s starting at the beginning of s and extending up to but not including position j.	type = "deciduous" type[:4]	 "deci"
x in s	True if x is found in s and False otherwise. x can be either a single letter or a possible substring of s.	'o' in "bob" 'd' in "bob"	True False
x not in s	False if x is found in s and True otherwise. x can be either a single letter or a possible substring of s.	"o" not in "bob" 'd' not in "bob"	False True
s.find(x)	If string s contains string x, this is the index of x's starting position in s. If s does not contain x, this is -1.	s = "abcde" s.find("cd") s.find("x")	 2 -1
s.lower()	Returns a string with the same characters as s, but all in lowercase.	s = "wHimseY" s.lower()	 "whimsey"
s.upper()	Returns a string with the same characters as s, but all in uppercase.	s = "wHimseY" s.upper()	 "WHIMSEY"
s.islower()	Returns True if s has at least one alphabetic letter and all of the alphabetic letters in s are lowercase.	s = "bob" s.islower()	 True
s.isupper()	Returns True if s has at least one alphabetic letter and all of the alphabetic letters in s are uppercase.	s = "OH!" s.isupper()	 True

Strings in Python can't be changed once they are created. For this reason strings are called *immutable*. This doesn't mean that a string variable can't be changed; it is only the string itself that cannot be altered. In this regard Python strings are no different from numbers: you cannot change the number 23 to be a different number. Some other languages have mutable strings. For example, in Java if `s` is the string `"picket"` and we execute a statement `s[1] = 'o'`, this changes string `s` to `"pocket"`. We can get the same effect in Python from the code:

```
s =   "picket"
s = s[:1] + 'o' + s[2:]
```

Note that rather than change a string, in Python we usually reconstruct it and assign the new string to the old string variable.

The string comparison operators use the same symbols as arithmetic comparisons. Strings are ordered by the usual dictionary (alphabetical) ordering, where a standard table of characters, called the *ASCII* table, is used to determine the ordering of individual characters. See the optional section on string encodings, below, for more details about this table. Here is a table of the comparison operators:

Symbol	Meaning	Example	Result
$<$	Less than	"*bike*" $<$ "*car*"	True
$>$	Greater than	"*bike*" $>$ "*car*"	False
$<=$	Less than or equal to	"*bike*" $<=$ "*car*"	True
		"*car*" $<=$ "*car*"	True
$>=$	Greater than or equal to	"*bike*" $>=$ "*car*"	False
		"*car*" $>=$ "*car*"	True
$==$	Equal to	"*bike*" $==$ "*car*"	False
	(comparison, not assignment)	"*car*" $==$ "*car*"	True
$!=$	Not equal to	"*bike*" $!=$ "*car*"	True
		"*car*" $!=$ "*car*"	False

String formatting

It is often the case that you want to print a line that is composed of text plus values that are contained in one or more variables. For example, an averaging program might have a variable average that contains the average, and a variable count that holds information about the number of item that went into this average. We might want to print all of this information on one line, such as:

```
7 items were found, with average 16.2.
```

To do this with string concatenation we would need to convert the numeric variables into strings:

```
print(str(count)+"items were found, with average "+str(average)+".")
```

This works, but it is hard to read and easy to get wrong. String formatting is a technique Python provides to simplify expressions like this. It is particularly

useful for print statements, but it can be used anywhere you need to make a string out of values contained in variables.

A formatted string consists of two parts separated by a percent sign:

```
pattern % (values)
```

The pattern can have percent-fields that act as placeholders. There are three such fields:

%d	is a placeholder for an integer value.
%f	is a placeholder for a floating point value.
%s	is a placeholder for a string.

For each percent-field in the pattern, you need to give a corresponding value. The values are a list, separated by commas, inside parentheses, such as (count, average) If you have only one value you may omit the parentheses.

The averaging example above would be expressed as follows with formatted strings:

```
print("%d items were found, with average %f."%(count,average))
```

This is easier to read, and easier to write correctly.

Formatted strings have another advantage. You are allowed to supply a field-width for each percent-field. This is very useful if you are trying to print data in a table so that it comes out in columns. If, for example, you have numbers that might need anywhere from 1 to 4 digits, printing them with a fieldwidth of 4 or more will guarantee that they always occupy the same amount of space in the output line.

For strings and integers the fieldwidth is a number that comes between the %-symbols and the s or d symbol. If the fieldwidth is positive, smaller values are right-justified; if it is negative they are left-justified. For example "%5d" means to print the integer value using at least 5 spaces, and to put the value at the right edge of this field. "%-5d" means to make the fieldwidth 5, and to put the value at the left edge of this field. With floats the fieldwidth is a bit more complex. "%w.df" means to use a total fieldwidth of w, including d places after the decimal point. For all of these fieldwidths, an omitted fieldwidth defaults to 1.

Altogether, our averaging example would be formatted:

```
print( "%d items were found, with average %.2f."%(count,average))
```

This means to print count using no more spaces than necessary, but to use 2 decimal places for average.

Here is another example, this time a complete program. We want to read in a person's name and age, and then print them out on one line. Program 2.5.1 does this. Note that we use **eval()** after the numeric input. Note also the use of formatted strings in the print statement. name is a string and age is an integer, so we use formats %s and %d formats for them.

```
# This reads a person's name and age,
# then prints them on one line

def main():
    name = input( "Enter a name: " )
    age = eval( input("Enter this person's age: ") )
    print( "%s is %d years old." % (name, age))

main()
```

Program 2.5.1: Names and Ages

Strings and encodings (optional)

Internally, all data inside a computer is represented by sequences of 0's and 1's. A given sequence of bits could represent either a number or a string, depending on how it is interpreted. Python lets us move easily between these two interpretations. You don't usually need to do this the mechanisms the language provides for manipulating text usually make it unnecessary to do this. In case you are curious, here are some details about how strings are encoded.

There are two built-in functions for working with the numeric encodings of strings:

ord(c)	gives the numeric value of c, which must be a single character: a string of length 1.
chr(n)	gives the character corresponding to integer n.

For example, ord('A') is 65 and ord('a') is 97. Python uses a standard character encoding called the *ASCII* character set. "ASCII" stands for "American Standard Code for Information Interchange". The ASCII encoding goes back to the early 1960's and has been the standard representation for text for many years. The first 31 characters in this set are non-printing "control" characters for controlling line printers and other output devices. In the table below we list all of the printing ASCII characters.

The ASCII Character Set											
x	**chr(x)**	**x**	**chr(x)**	**x**	**chr(x)**	**x**	**chr(x)**	**x**	**chr(x)**	**x**	**chr(x)**
32	space	51	3	70	F	89	Y	108	l
33	!	52	4	71	G	90	Z	109	m
34	”	53	5	72	H	91	[	110	n
35	#	54	6	73	I	92	\	111	o
36	$	55	7	74	J	93	]	112	p
37	%	56	8	75	K	94	^	113	q
38	&	57	9	76	L	95	_	114	r
39	’	58	:	77	M	96	‘	115	s
40	(	59	;	78	N	97	a	116	t
41	)	60	¡	79	O	98	b	117	u
42	*	61	=	80	P	99	c	118	v
43	+	62	¿	81	Q	100	d	119	w
44	,	63	?	82	R	101	e	120	x
45	-	64	@	83	S	102	f	121	y
46	.	65	A	84	T	103	g	122	z
47	/	66	B	85	U	104	h	123	{
48	0	67	C	86	V	105	i	124	|
49	1	68	D	87	W	106	j	125	}
50	2	69	E	88	X	107	k	126	~

You can see that this character set has some convenient characteristics: the lower-case letters are all contiguous and the upper-case letters are too, though the upper-case letters and the lower-case ones do not appear together. In fact, all of the upper-case letters come before all of the lower-case ones. This ordering is used when we alphabetize words in Python: unless we do extra work to avoid this, anything that starts with an upper-case letter will appear before anything that starts with a lower-case one.

Python uses ASCII encodings for strings when it is possible, because each of the ASCII characters can be encoded in one byte (8 bits) of data. However, Python also implements an extension of ASCII called /em Unicode that uses two bytes per character. This allows for about 65,000 different characters. For example, chr(9924) is a cute picture of a snowman while chr(9824) is the ”spade” character from a deck of playing cards. If you run the code

```
for i in range(0, 20000):
    print( "%d: %s" %(i, chr(i)))
```

you can see many of the options for this extended character set.