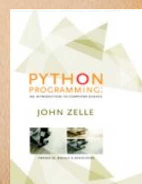
Python Programming: An Introduction to Computer Science

Chapter 4

Computing with Strings



The String Data Type

- The most common use of personal computers is word processing.
- Text is represented in programs by the *string* data type.
- A string is a sequence of characters enclosed within quotation marks (") or apostrophes (') or (“””).

Examples

```
>>> str1="Hello"
>>> str2='spam'
>>> print str1, str2
Hello spam
>>> type(str1)
<type 'str'>
>>> type(str2)
<type 'str'>
```


Newline string examples

To create a string with a *newline* (“return”) in it, you can add a *newline* character \n .You can think of *newline* as the character produced when you press the <Enter> key

```
>>>print “The newspaper is \n at the door”
The newspaper is
 at the door
>>> print “Mary: John? \nJohn: Yes \nMary: Your car is on fire”
Mary: John?
John: Yes
Mary: Your car is on fire
```

Line continuation examples

To create a very long string you may want to define it on multiple lines, for this you can use the line continuation character “\”

```
>>>hello = "This is a rather long string containing\n\
several lines of text just as you would do in C.\n\
    Note that whitespace at the beginning of the line is\
significant."
>>>print hello
This is a rather long string containing
several lines of text just as you would do in C.
    Note that whitespace at the beginning of the line is significant.
```


“”” triple-quotes examples

Triple-quotes “”” tells Python to include newline characters and you do not need the line continuation characters. This is useful if you have a large block of formatted text, just type it as you want it to look.

```
>>> str1=“”"
    Usage: thingy [OPTIONS]
        -h                    Display this usage message
        -H hostname        Hostname to connect to
"""
>>> print str1
  Usage: thingy [OPTIONS]
      -h                    Display this usage message
      -H hostname       Hostname to connect to
```


Why input doesn’t work

```
>>> firstName = input("Please enter your name: ")
Please enter your name: John

Traceback (most recent call last):
  File "<pyshell#12>", line 1, in -toplevel-
    firstName = input("Please enter your name: ")
  File "<string>", line 0, in -toplevel-
NameError: name 'John' is not defined
```

- What happened?

Why Input Doesn’t Work (cont)

- The input statement is a delayed expression.
- When you enter a name, it’s doing the same thing as:
  firstName = John
- The way Python evaluates expressions is to look up the value of the variable John and store it in firstName.
- Since John didn’t have a value, we get a NameError.

Using quotes to avoid the problem

- One way to fix this is to enter your string input with quotes around it:

```
>>> firstName = input("Please enter your name: ")
Please enter your name: "John"
>>> print "Hello", firstName
Hello John
```

- Even though this works, this is cumbersome!

Using raw_input to avoid the problem

- There is a better way to handle text – the raw_input function.
- raw_input is like input, but it doesn't evaluate the expression that the user enters.

```
>>> firstName = raw_input("Please enter your name: ")
Please enter your name: John
>>> print "Hello", firstName
Hello John
```


Accessing individual characters

- We can access the individual characters in a string through *indexing*.
- The positions in a string are numbered from the left, starting with 0.
- The general form is <string>[<expr>], where the value of expr determines which character is selected from the string.

Indexing example

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

```
>>> greet = "Hello Bob"
>>> greet[0]
'H'
>>> print greet[0], greet[2], greet[4]
H l o
>>> x = 8
>>> print greet[x - 2]
B
```

Indexing example - from the right

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

- In a string of *n* characters, the last character is at position *n-1* since we start counting with 0.
- We can index from the right side using negative indexes.

```
>>> greet[-1]
'b'
>>> greet[-3]
'B'
```


What about a substring? Slicing a string

- Slicing:
  <string>[<start>:<end>]
- start and end should both be ints
- The slice contains the substring beginning at position start and runs up to **but doesn't include** the position end.

Slicing Example

H	e	l	l	o		B	o	b

0 1 2 3 4 5 6 7 8 9

```
>>> greet[0:3]
'Hel'
>>> greet[5:9]
' Bob'
>>> greet[:5]
'Hello'
>>> greet[5:]
' Bob'
>>> greet[:]
'Hello Bob'
```

Hint: When slicing it helps to think of the slide indexes between the characters, then 0:3 is very clear

String Operators

Operator	Meaning
+	Concatenation
*	Repetition
<string>[]	Indexing
<string>[:]	Slicing
len(<string>)	Length
for <var> in <string>	Iteration through characters

Concatenation (str + str)

- Concatenation adds two strings together to for another string

```
>>> "spam" + "eggs"
'spameggs'
>>> x = "Dan"
>>> y = "WasHere"
>>> z = x+y
>>> print z
DanWasHere
```


Repetition (str * number)

- Repetition creates a string by copying the string X times

```
>>> "spam" * 3
'spamspamspam'
>>> x = 10
>>> print "o" * x
oooooooooo
>>> myStr = 4 * "abc"
>>> print myStr
abcabcabc
```


len(str) returns string's length

- len(str) returns the number of characters in the string (its "length")

```
>>> len("spam")
4
>>> y = len( "spam" * 3)
```

WHAT IS Y?

Iterating through characters

- For <var> in <string> this loops through each character in the string, assigning the value to <var>

```
>>> for ch in "Spam!":
    print ch,

S p a m !
```

Example: Reversing a String

- Using these functions how would you reverse a string (spam --> maps)?

Pseudocode

What is it?

Python Code

What is it?

Simple String Processing

- Usernames on a computer system
  - First initial, first seven characters of last name

```
# get user's first and last names
first = raw_input("Please enter your first name (all lowercase): ")
last = raw_input("Please enter your last name (all lowercase): ")

# concatenate first initial with 7 chars of last name
```

How?

Simple String Processing

```
>>>
Please enter your first name (all lowercase): john
Please enter your last name (all lowercase): doe
uname =  jdoe

>>>
Please enter your first name (all lowercase): donna
Please enter your last name (all lowercase): rostenkowski
uname =  drostenk
```


Simple String Processing

- Another use – converting an int that stands for the month into the three letter abbreviation for that month.
- Store all the names in one big string: "JanFebMarAprMayJunJulAugSepOctNovDec"
- Use the month number as an index for slicing this string:

Simple String Processing

Month	Number	Position
Jan	1	0
Feb	2	3
Mar	3	6
Apr	4	9

- To get the correct position, subtract one from the month number and multiply by three

Simple String Processing

```
# month.py - A program to print the abbreviation of a month, given its number
def main():

    # months is used as a lookup table
    months = "JanFebMarAprMayJunJulAugSepOctNovDec"

    n = input("Enter a month number (1-12): ")

    # compute starting position of month n in months
    pos = (n-1) * 3

    # Grab the appropriate slice from months
    monthAbbrev = months[pos:pos+3]

    # print the result
    print "The month abbreviation is", monthAbbrev + "."

main()
```


Simple String Processing

```
>>> main()
Enter a month number (1-12): 1
The month abbreviation is Jan.
>>> main()
Enter a month number (1-12): 12
The month abbreviation is Dec.
```

- One weakness – this method only works where the potential outputs all have the same length.
- How could you handle spelling out the months?

Strings, Lists, and Sequences

- It turns out that strings are really a special kind of *sequence*, so these operations also apply to sequences!

```
>>> [1,2] + [3,4]
[1, 2, 3, 4]
>>> [1,2]*3
[1, 2, 1, 2, 1, 2]
>>> grades = ['A', 'B', 'C', 'D', 'F']
>>> grades[0]
'A'
>>> grades[2:4]
['C', 'D']
>>> len(grades)
5
```

Intro to Lists

- Strings are always sequences of characters, but *lists* can be sequences of arbitrary values.
- Lists can have numbers, strings, or both!

```
myList = [1, "Spam ", 4, "U"]
```


Lists as a lookup table

- We can use the idea of a list to make our previous month program even simpler!
- We change the lookup table for months to a list:

```
months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
"Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]
```


Lists as a lookup table

- To get the months out of the sequence, do this:

```
monthAbbrev = months[n-1]
```

Rather than this:

```
monthAbbrev = months[pos:pos+3]
```


Strings, Lists, and Sequences

```
# month2.py
#  A program to print the month name, given it's
    number.
#  This version uses a list as a lookup table.

def main():

  # months is a list used as a lookup table
  months = ["Jan", "Feb", "Mar", "Apr", "May",
    "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]

  n = input("Enter a month number (1-12): ")

  print "The month abbreviation is", months[n-1] +
    "."

main()
```

Note that the months line overlaps a line. Python knows that the expression isn't complete until the closing] is encountered.

Strings, Lists, and Sequences

```
# month2.py
#  A program to print the month name, given it's
     number.
#  This version uses a list as a lookup table.

def main():

   # months is a list used as a lookup table
   months = ["Jan", "Feb", "Mar", "Apr", "May",
     "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]

   n = input("Enter a month number (1-12): ")

   print "The month abbreviation is", months[n-1] +
     "."

main()
```

Since the list is indexed starting from 0, the *n-1* calculation is straight-forward enough to put in the print statement without needing a separate step.

Strings, Lists, and Sequences

- This version of the program is easy to extend to print out the whole month name rather than an abbreviation!

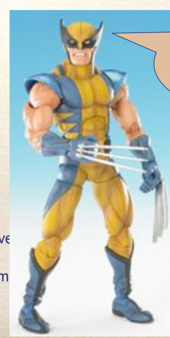
```
 months = ["January", "February", "March", "April", "May",
"June", "July", "August", "September", "October", "November",
"December"]
```


Strings, Lists, and Sequences

- Lists are *mutable*, meaning they can be changed. Strings can **not** be changed.

```
>>> myList = [34, 26, 15, 10]
>>> myList[2]
15
>>> myList[2] = 0
>>> myList
[34, 26, 0, 10]
>>> myString = "Hello World"
>>> myString[2]
'l'
>>> myString[2] = "p"

Traceback (most recent call last):
  File "<pyshell#16>", line 1, in -topleve
    myString[2] = "p"
TypeError: object doesn't support item
```



What do you mean no mutations for Strings?

List methods

- Lists have some other methods you can use
- Why type what you can reference… lets just go here: http://docs.python.org/tut/node7.html

List example

- Lets say you were trying to store the state of a tic-tac-toe board in a list. How would you represent this board:

	O	O
	X	
	X	

Thoughts? What would you say to someone over the phone to describe the board?

List example

- Label each “spot” with a number, then create a list with those numbers.

0	1	2
3	4	5
6	7	8

	O	O
	X	
	X	

boardState = [‘_‘,’O’,’O’,’_‘,’X’,’_‘,’_‘,’X’,’_‘]

String Library

- Just like there is a math library, there is a string library with many handy functions.
- One of these functions is called *split*. This function will split a string into substrings based on spaces.

```
>>> import string
>>> string.split("Hello string library!")
['Hello', 'string', 'library!']
```


Split example

- Split can be used on characters other than space, by supplying that character as a second parameter.

```
>>> string.split("32,24,25,57", ",")
['32', '24', '25', '57']
>>>
```

Converting Strings to Numbers

- How can we convert a string containing digits into a number?
- Python has a function called *eval* that takes any strings and evaluates it as if it were an expression.

```
>>> numStr = "500"
>>> eval(numStr)
500
>>> x = eval(raw_input("Enter a number "))
Enter a number 3.14
>>> print x
3.14
>>> type (x)
<type 'float'>
```


A String Formatting Example

Let's see if we can make a hang man game using strings.

How would you do it?

Input/Output as String Manipulation

- Often we will need to do some string operations to prepare our string data for output ("pretty it up")
- Let's say we want to enter a date in the format "05/24/2003" and output "May 24, 2003." How could we do that?

Input/Output as String Manipulation

- Input the date in mm/dd/yyyy format (dateStr)
- Split dateStr into month, day, and year strings
- Convert the month string into a month number
- Use the month number to lookup the month name
- Create a new date string in the form "Month Day, Year"
- Output the new date string

Input/Output as String Manipulation

- The first two lines are easily implemented!
  dateStr = raw_input("Enter a date (mm/dd/yyyy): ")
  monthStr, dayStr, yearStr = string.split(dateStr, "/")
- The date is input as a string, and then "unpacked" into the three variables by splitting it at the slashes using simultaneous assignment.

Coming up: Input/Output as String Manipulation 45

Input/Output as String Manipulation

- Next step: Convert monthStr into a number
- We can use the *eval* function on monthStr to convert "05", for example, into the integer 5. (eval("05") = 5)
- Another conversion technique would be to use the *int* function. (int("05") = 5)

Coming up: Input/Output as String Manipulation 46

Input/Output as String Manipulation

- There's one "gotcha" – leading zeros.
- >>> int("05")
  5
  >>> eval("05")
  5
- >>> int("023")
  23
  >>> eval("023")
  19
- What's going on??? Int seems to ignore leading zeroes, but what about eval?

Coming up: Eval and Octal Numbers 47

Eval and Octal Numbers

- Python allows int literals to be expressed in other number systems than base 10! If an int starts with a 0, Python treats it as a base 8 (octal) number.
- $023_8 = 2*8 + 3*1 = 19_{10}$
- OK, that's interesting, but why support other number systems?

Coming up: The Rule: Use int to convert numbers 48

The Rule: Use int to convert numbers

- Computers use base 2 (binary). Octal is a convenient way to represent binary numbers.
- If this makes your brain hurt, just remember to use int rather than eval when converting strings to numbers when there might be leading zeros.

Input/Output as String Manipulation

months = [“January”, “February”, …, “December”]
monthStr = months[int(monthStr) – 1]

- Remember that since we start counting at 0, we need to subtract one from the month.
- Now let’s concatenate the output string together!

Input/Output as String Manipulation

print “The converted date is:”, monthStr, dayStr+”,”, yearStr

- Notice how the comma is appended to dayStr with concatenation!
- >>> main()
  Enter a date (mm/dd/yyyy): 01/23/2004
  The converted date is: January 23, 2004

Converting a number to a string

- Sometimes we want to convert a number into a string.
- We can use the *str* function!

```
>>> str(500)
'500'
>>> value = 3.14
>>> str(value)
'3.14'
>>> print "The value is", str(value) + "."
The value is 3.14.
```

Converting a number to a string

- If value is a string, we can concatenate a period onto the end of it.
- If value is an int, what happens?

```
>>> value = 3.14
>>> print "The value is", value + "."
The value is

Traceback (most recent call last):
  File "<pyshell#10>", line 1, in -toplevel-
    print "The value is", value + "."
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```


Converting a number to a string

- If value is an int, Python thinks the + is a mathematical operation, not concatenation, and "." is not a number!

Conversion Operations

- We now have a complete set of type conversion operations:

Function	Meaning
float(<expr>)	Convert expr to a floating point value
int(<expr>)	Convert expr to an integer value
long(<expr>)	Convert expr to a long integer value
str(<expr>)	Return a string representation of expr
eval(<string>)	Evaluate string as an expression

String Formatting using templates

- <template-string> % (<values>)
- % within the template-string mark "slots" into which the values are inserted.
- There must be one slot per value.
- Each slot has a *format specifier* that tells Python how the value for the slot should appear.

String Formatting

print "The total value of your change is $%0.2f “ % (total)

- The template contains a single specifier: %0.2f
- The value of total will be inserted into the template in place of the specifier.
- The specifier tells us this is a floating point number (f) with two decimal places (.2)

String Formatting

- The formatting specifier has the form: %<width>.<precision><type-char>
- Type-char can be **d**ecimal, **f**loat, **s**tring (decimal is base-10 ints)
- <width> and <precision> are optional.
- <width> tells us how many spaces to use to display the value. 0 means to use as much space as necessary.

String Formatting

- If you don’t give it enough space using <width>, Python will expand the space until the result fits.
- <precision> is used with floating point numbers to indicate the number of places to display after the decimal.
- %0.2f means to use as much space as necessary and two decimal places to display a floating point number.

String Formatting

```
>>> "Hello %s %s, you may have already won $%d" % ("Mr.", "Smith",
    10000)
'Hello Mr. Smith, you may have already won $10000’

>>> 'This int, %5d, was placed in a field of width 5' % (7)
'This int,     7, was placed in a field of width 5'

>>> 'This int, %10d, was placed in a field of witdh 10' % (10)
'This int,         10, was placed in a field of witdh 10'

>>> 'This int, %10d, was placed in a field of width 10' % (7)
'This int,          7, was placed in a field of width 10'
```

String Formatting

```
>>> 'This float, %10.5f, has width 10 and precision 5.' % (3.1415926)
'This float,    3.14159, has width 10 and precision 5.'

>>> 'This float, %0.5f, has width 0 and precision 5.' % (3.1415926)
'This float, 3.14159, has width 0 and precision 5.'

>>> 'Compare %f and %0.20f' % (3.14, 3.14)
'Compare 3.140000 and 3.14000000000000010000'
```


String Formatting

- If the width is wider than needed, the value is right-justified by default. You can left-justify using a negative width (%-10.5f)

Another way to format..

```
import string
x = “Joe”
x.ljust(10)          # Left justify this string in a 10 char field
x.rjust(10)          # Right justify this string in a 10 char field
x.center(10)         # Center this string in a 10 char field

>>> x = "Joe"
>>> x.ljust(10)
'Joe       '
>>> x.rjust(10)
'       Joe'
>>> x.center(10)
'   Joe    '
>>>
```


Example: Making Change…

- Assume you are working at cash register and you need a program to calculate the number of coins to give out for a given number. For example, if the price is $17.23 and someone give you a $20, you should return:
- 2 dollars
- 3 quarters
- 2 pennies

Example: Making Change…

- Ask for cost
- Ask for payment amount
- Compute number of dollars
- Compute number of quarters, dimes, nickels, pennies
- Output (nicely) to the user

Example: Making Change…

- Ask for cost
- Ask for amount input

cost, payment = input(“What is the cost and payment ->”)

Detailed Pseudocode: dollars

Determine number of dollars

determine change (payment - cost)
remove decimal value
return that as the number of dollars

Detailed Pseudocode: coins

Determine number of coins

determine change (payment - cost - dollars)
figure out how many quarters you can use
remove that value from the change
figure out how many dimes you can use
subtract that value from the change…

Example: Making change…

- On to computechangeSkeleton.py

Example: Verifying input…

- If we provide a menu to the user with numbers, how can we validate that they enter a valid number? Let's try…
- First, to the string documentation…
- Next up: Pseudocode

Pseudocode: Verifying input…

1. ask user the question
2. check if answer is all digits
3. if so
   - convert the answer to an int
   - return the int to the user
4. else
   - print an error
   - go back to step 2

Example: Verifying input…

On to inputverification.py…