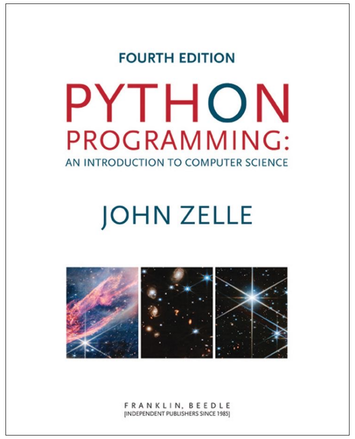
Python Programming: An Introduction to Computer Science



Chapter 8
Strings

Objectives

- To understand the string data type and how strings are represented in the computer.
- To become familiar with various operations that can be performed on strings through built-in functions and string methods.

Objectives

- To understand the basic idea of sequences and indexing as they apply to Python strings and lists.
- To be able to apply string formatting to produce attractive, informative program output.
- To understand basic concepts of cryptography.
- To be able to understand and write programs that process textual information.

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 (').

The String Data Type

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

The String Data Type

- Getting a string as input

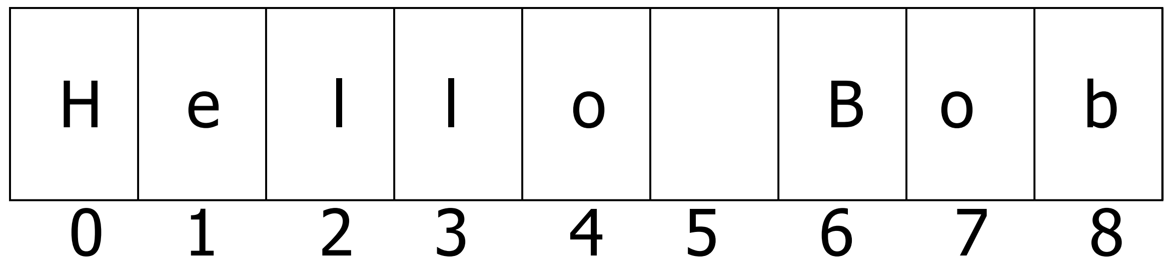
```
>>> firstName = input("Please enter your name: ")
Please enter your name: John
>>> print("Hello", firstName)
Hello John
```

- Notice how we saved the user’s name with a variable to print the name back out again.

The String Data Type

- 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.

The String Data Type



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

The Python shell shows us the value of strings by putting them in single quotes; when we print the string, Python does not put any quotes around the sequence of characters.

The String Data Type

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'
```

The String Data Type

- Indexing returns a string containing a single character from a larger string.
- We can also access a contiguous sequence of characters, called a *substring*, through a process called *slicing*.

The String Data Type

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`.

The String Data Type

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

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

The String Data Type

- If either expression is missing, then the start or the end of the string are used.
- Can we put two strings together into a longer string?
- *Concatenation* "glues" two strings together (+)
- *Repetition* builds up a string by multiple concatenations of a string with itself (*)

The String Data Type

- The function *len* will return the length of a string.

```
>>> "spam" + "eggs"
'spameggs'
>>> "Spam" + "And" + "Eggs"
'SpamAndEggs'
>>> 3 * "spam"
'spamspamspam'
>>> "spam" * 5
'spamspamspamspamspam'
>>> (3 * "spam") + ("eggs" * 5)
'spamspamspameggseggseggseggseggs'
```

The String Data Type

```
>>> len("spam")
4
>>> for ch in "Spam!":
        print (ch, end=" ")

S p a m !
```

The String Data Type

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

Simple String Processing

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

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

# concatenate first initial with 7 chars of last name
uname = first[0] + last[:7]
```

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

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 = int(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 + ".")
```

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?

String Representation

- Inside the computer, strings are represented as sequences of 1's and 0's, just like numbers.
- A string is stored as a sequence of binary numbers, one number per character.
- It doesn't matter what value is assigned as long as it's done consistently.

String Representation

- In the early days of computers, each manufacturer used their own encoding of numbers for characters.
- ASCII system (American Standard Code for Information Interchange) uses numbers between 0 and 127.
- Python supports Unicode (100,000+ characters)

String Representation

- The *ord* function returns the numeric (ordinal) code of a single character.
- The *chr* function converts a numeric code to the corresponding character.

```
>>> ord("A")
65
>>> ord("a")
97
>>> chr(97)
'a'
>>> chr(65)
'A'
```

String Representation

The smallest addressable memory unit is generally 8 bits, called a byte.
8 bits can be used to encode up to 256 values, more than enough for ASCII.
Unicode is different in that it uses various encoding schemes to pack Unicode characters into sequences of bytes.
The most common encoding used for this is called UTF-8. UTF-8 uses between one (for Latin alphabets) and four bytes for some of the more exotic characters.

Programming an Encoder

- Using `ord` and `char` we can convert a string into and out of numeric form.
- The encoding algorithm is simple:

```
get the message to encode
for each character in the message:
  print the letter number of the character
```

- A `for` loop iterates over a sequence of objects, so the for loop looks like:

```
for ch in <string>
```

Programming an Encoder

```
# text2numbers.py
#     A program to convert a textual message into a sequence of
#         numbers, utlilizing the underlying Unicode encoding.

def main():
    print("This program converts a textual message into a sequence")
    print ("of numbers representing the Unicode encoding of the message.\n")

    # Get the message to encode
    message = input("Please enter the message to encode: ")

    print("\nHere are the Unicode codes:")

    # Loop through the message and print out the Unicode values
    for ch in message:
        print(ord(ch),  end=" ")

    print()  # blank line before prompt
```

Programming a Decoder

- We now have a program to convert messages into a type of “code”, but it would be nice to have a program that could decode the message!
- The outline for a decoder:

```
get the sequence of numbers to decode
message = ""
for each number in the input:
   convert the number to the appropriate character
   add the character to the end of the message
print message
```

Programming a Decoder

- The variable *message* is an accumulator variable, initially set to the *empty string*, the string with no characters ("").
- Each time through the loop, a number from the input is converted to the appropriate character and appended to the end of the accumulator.

Programming a Decoder

- How do we get the sequence of numbers to decode? We don’t know how many numbers there will be!
- Read the input as a single string, then split it apart into substrings, each of which represents one number.
- Iterate through the list of smaller strings, convert each into a number and use that number to produce the corresponding Unicode character.

Programming a Decoder

The new algorithm

```
get the sequence of numbers as a string, inString
split inString into a sequence of smaller strings
message = ""
for each of the smaller strings:
   change the string of digits into the number it represents
   append the Unicode character for that number to message
print message
```

Programming a Decoder

- Strings are objects and have useful methods associated with them
- One of these methods is *split*. This will split a string into substrings based on spaces.

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

Programming a Decoder

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

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

Programming a Decoder

```
# numbers2text.py
#     A program to convert a sequence of Unicode numbers into
#         a string of text.

def main():
    print ("This program converts a sequence of Unicode numbers into")
    print ("the string of text that it represents.\n")

    # Get the message to encode
    inString = input("Please enter the Unicode-encoded message: ")

    # Loop through each substring and build Unicde message
    message = ""
    for numStr in inString.split():
        # convert the (sub)string to a number
        codeNum = int(numStr)
        # append character to message
        message = message + chr(codeNum)

    print("\nThe decoded message is:", message)
```

Programming a Decoder

- The `split` function produces a list of substrings. `numString` gets each successive substring.
- Each time through the loop, the next substring is converted to the appropriate Unicode character and appended to the end of the accumulator, `message`.

Programming a Decoder

```
--------------------------------------------------------------------------
This program converts a textual message into a sequence
of numbers representing the Unicode encoding of the message.

Please enter the message to encode: CS120 is fun!

Here are the Unicode codes:
67 83 49 50 48 32 105 115 32 102 117 110 33

---------------------------------------------------------------------------
This program converts a sequence of Unicode numbers into
the string of text that it represents.

Please enter the ASCII-encoded message: 67 83 49 50 48 32 105 115 32 102 117 110 33
The decoded message is: CS120 is fun!
```

From Encoding to Encryption

- The process of encoding information for the purpose of keeping it secret or transmitting it privately is called *encryption*.
- *Cryptography* is the study of encryption methods.
- Encryption is used when transmitting credit card and other personal information to a web site.

From Encoding to Encryption

- Strings are represented as a sort of encoding problem, where each character in the string is represented as a number that's stored in the computer.
- The code that is the mapping between character and number is an industry standard, so it's not "secret".

From Encoding to Encryption

- The encoding/decoding programs we wrote use a *substitution cipher*, where each character of the original message, known as the *plaintext*, is replaced by a corresponding symbol in the *cipher alphabet*.
- The resulting code is known as the *ciphertext.*

From Encoding to Encryption

- This type of code is relatively easy to break.
- Each letter is always encoded with the same symbol, so using statistical analysis on the frequency of the letters and trial and error, the original message can be determined.

From Encoding to Encryption

- Modern encryption converts messages into numbers.
- Sophisticated mathematical formulas convert these numbers into new numbers – usually this transformation consists of combining the message with another value called the "*key*"

From Encoding to Encryption

- To decrypt the message, the receiving end needs an appropriate key so the encoding can be reversed.
- In a *private key* (or *shared key*) system the same key is used for encrypting and decrypting messages. Everyone you know would need a copy of this key to communicate with you, but it needs to be kept a secret.

From Encoding to Encryption

- In *public key* encryption, there are separate keys for encrypting and decrypting the message.
- In public key systems, the encryption key is made publicly available, while the decryption key is kept private.
- Anyone with the public key can send a message, but only the person who holds the private key (decryption key) can decrypt it.

More String Methods

- There are a number of other string methods. Try them all!
  - `s.capitalize()` – Copy of s with only the first character capitalized
  - `s.title()` – Copy of s; first character of each word capitalized
  - `s.center(width)` – Center s in a field of given width

More String Methods

- `s.count(sub)` – Count the number of occurrences of sub in s
- `s.find(sub)` – Find the first position where sub occurs in s
- `s.join(list)` – Concatenate list of strings into one large string using s as separator.
- `s.ljust(width)` – Like center, but s is left-justified

More String Methods

- `s.lower()` – Copy of s in all lowercase letters
- `s.lstrip()` – Copy of s with leading whitespace removed
- `s.replace(oldsub, newsub)` – Replace occurrences of oldsub in s with newsub
- `s.rfind(sub)` – Like find, but returns the right-most position
- `s.rjust(width)` – Like center, but s is right-justified

More String Methods

- `s.rstrip()` – Copy of s with trailing whitespace removed
- `s.split()` – Split s into a list of substrings
- `s.upper()` – Copy of s; all characters converted to uppercase

Better Change Counter

- With what we know now about floating point numbers, we might be uneasy about using them in a money situation.
- One way around this problem is to keep track of money in cents using an int or long int, and convert it into dollars and cents when output.

Better Change Counter

If total is a value in cents (an int),

```
dollars = total//100
```

- `cents = total%100`
- Cents is printed using width 0>2 to right-justify it with leading 0s (if necessary) into a field of width 2.
- Thus 5 cents becomes '05'

Better Change Counter

```
# change2.py
#   A program to calculate the value of some change in dollars.
#   This version represents the total cash in cents.

def main():
    print ("Change Counter\n")

    print ("Please enter the count of each coin type.")
    quarters = int(input("Quarters: "))
    dimes = int(input("Dimes: "))
    nickels = int(input("Nickels: "))
    pennies = int(input("Pennies: "))
    total = quarters * 25 + dimes * 10 + nickels * 5 + pennies
    dollars, cents = divmod(total, 100)

    print (f"The total value of your change is ${dollars}.{cents:0>2}.")
```

Better Change Counter

```
>>> main()
Change Counter

Please enter the count of each coin type.
Quarters: 0
Dimes: 0
Nickels: 0
Pennies: 1

The total value of your change is $0.01
```

```
>>> main()
Change Counter

Please enter the count of each coin type.
Quarters: 12
Dimes: 1
Nickels: 0
Pennies: 4

The total value of your change is $3.14
```