# Working with Strings

# Outline

1. The String Type
2. Unicode
3. String Operations
4. A Preview of Functions and Methods
5. Formatted Output for Strings
6. Control and Strings
7. Working with Strings

# The String Type

# Sequence of characters

- We've talked about strings being a sequence of characters.
- A string is indicated between `' '` or `" "`
- The exact sequence of characters is maintained
- It is also a collection
- Create the object with assignment or *str* constructor

# The Triple Quote String

- Triple quotes preserve both the vertical and horizontal formatting of the string
- Allows you to type tables, paragraphs, whatever and preserve the formatting

```
'''this is
a test
Today'''
```

- Also used for multi-line comments

# Non-printing Characters

If inserted directly, are preceded by a backslash (the \ character)

- new line `'\n'`
- tab `'\t'`

# Strings

- Can use single or double quotes:

```
S = "spam"
s = 'spam'
```

- Just don't mix them

```
my_str = 'hi mom"
```
$\Rightarrow$ ERROR

- Inserting an apostrophe:

```
A = "knight's"    # mix up the quotes
B = 'knight\'s'   # escape single quote
```

# String Representation

- Every character is "mapped" (associated) with an integer
- UTF-8, subset of Unicode, is such a mapping
- the function `ord()` takes a character and returns its UTF-8 integer value, `chr()` takes an integer and returns the UTF-8 character.

```
>>> ord('a')
97
>>> ord('?')
63
>>> ord('\n')
10
>>> chr(10)
'\n'
>>> chr(63)
'?'
>>> chr(97)
'a'
>>>
```

# Subset of UTF-8

| Char | Dec | Char | Dec | Char | Dec |
|---|---|---|---|---|---|
| SP | 32 | @ | 64 | ` | 96 |
| ! | 33 | A | 65 | a | 97 |
| " | 34 | B | 66 | b | 98 |
| # | 35 | C | 67 | c | 99 |
| $ | 36 | D | 68 | d | 100 |
| % | 37 | E | 69 | e | 101 |
| & | 38 | F | 70 | f | 102 |
| ' | 39 | G | 71 | g | 103 |
| ( | 40 | H | 72 | h | 104 |
| ) | 41 | I | 73 | i | 105 |
| * | 42 | J | 74 | j | 106 |
| + | 43 | K | 75 | k | 107 |
| , | 44 | L | 76 | l | 108 |
| - | 45 | M | 77 | m | 109 |
| . | 46 | N | 78 | n | 110 |
| / | 47 | O | 79 | o | 111 |
| 0 | 48 | P | 80 | p | 112 |
| 1 | 49 | Q | 81 | q | 113 |
| 2 | 50 | R | 82 | r | 114 |
| 3 | 51 | S | 83 | s | 115 |
| 4 | 52 | T | 84 | t | 116 |

See Appendix D for the full set

# Unicode

In Unicode, every character is represented by an integer code point.

The code point is not necessarily the actual byte representation of the character; it is just the identifier for the particular character

The code point for letter `a` is the integer with hexadecimal value 0x0061

- Unicode conveniently uses a code point for ASCII characters that is equal to their ASCII code

With Unicode, we can write strings in

- english
- cyrillic

escape sequence `\u` indicates start of Unicode code point

```
>>> '\u0061'
'a'
>>> '\u0064\u0061d'
'dad'
>>>
'\u0409\u0443\u0431\u043e\u043c\u
0438\u0440'
'Љубомир'
>>> '\u4e16\u754c\u60a8\u597d!'
'世界您好!'
>>>
```

# String comparison, revisited

Unicode code points, being integers, give a natural ordering to all the characters representable in Unicode

Unicode was designed so that, for any pair of characters from the same alphabet, the one that is earlier in the alphabet will have a smaller Unicode code point.

```
>>> s1 = '\u0021'
>>> s1
'!'
>>> s2 = '\u0409'
>>> s2
'Љ'
>>> s1 < s2
True
>>>
```

# Unicode Transformation Format (UTF)

A Unicode string is a sequence of code points that are numbers from 0 to 0x10FFFF.

Unlike ASCII codes, Unicode code points are not what is stored in memory; the rule for translating a Unicode character or code point into a sequence of bytes is called an encoding.

There are several Unicode encodings: UTF-8, UTF-16, and UTF-32. UTF stands for Unicode Transformation Format.

- UTF-8 has become the preferred encoding for e-mail and web pages
- The default encoding when you write Python 3 programs is UTF-8.
- In UTF-8, every ASCII character has an encoding that is exactly the 8-bit ASCII encoding.

# The Index

- Because the elements of a string are a sequence, we can associate each element with an ***index***, a location in the sequence:
  - positive values count up from the left, beginning with index 0
  - negative values count down from the right, starting with -1

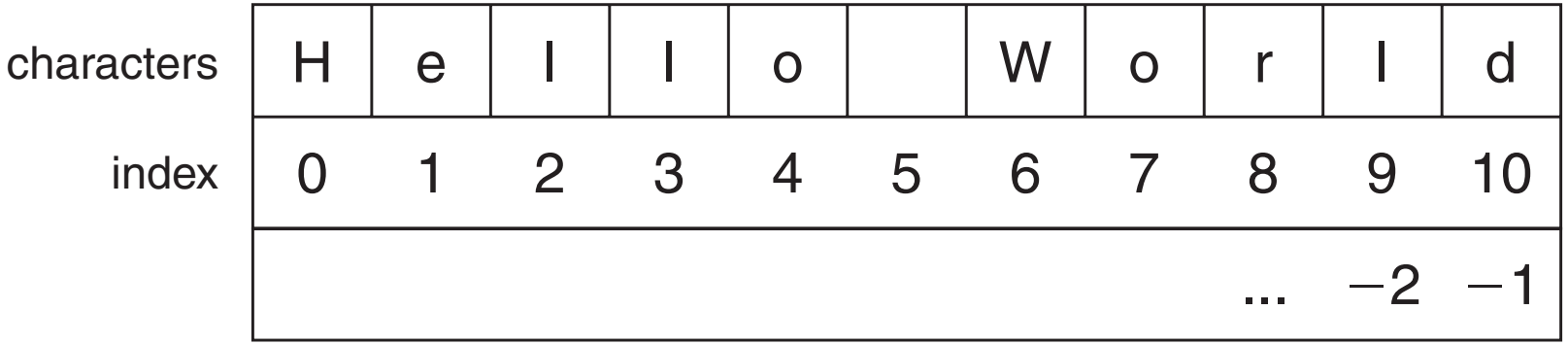



**FIGURE 4.1** The index values for the string `'Hello World'`.

# Accessing an element

A particular element of the string is accessed by the index of the element surrounded by square brackets [ ]

```
hello_str = 'Hello World'
print(hello_str[1])   => prints e
print(hello_str[-1])  => prints d
print(hello_str[11])  => ERROR
```

# Slicing, the rules

- slicing is the ability to select a subsequence of the overall sequence
- uses the syntax `[start : finish]`, where:
  - `start` is the index of where we start the subsequence
  - `finish` is the index of **one after** where we end the subsequence
- if either `start` or `finish` are not provided, it defaults to the beginning of the sequence for `start` and the end of the sequence for `finish`

# Half Open Range for Slices

- slicing uses what is called a half-open range
- the first index is included in the sequence
- the last index is one ***after*** what is included

```
helloString[6:10]
```

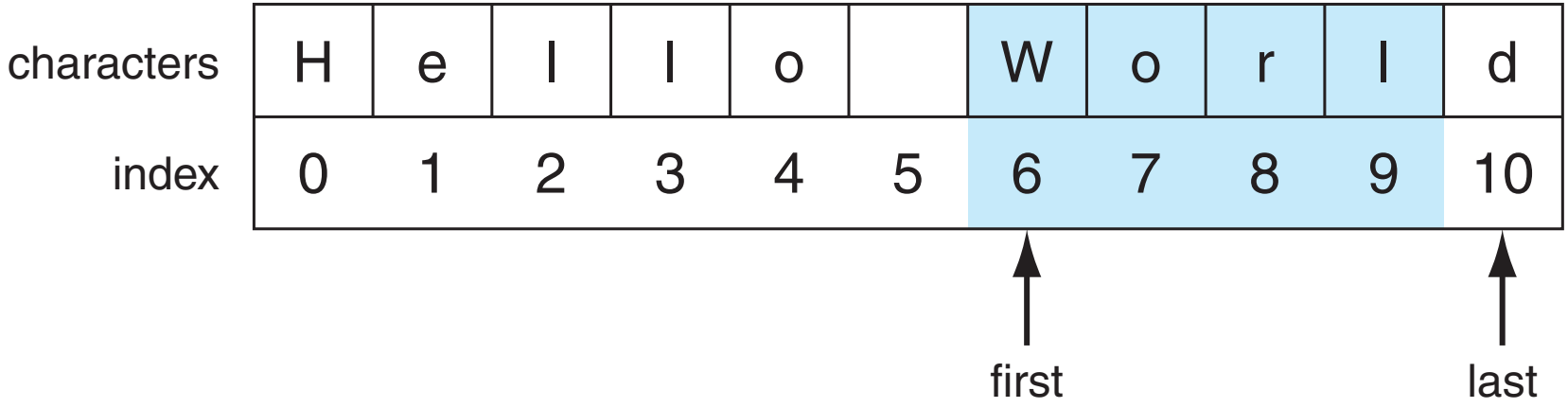
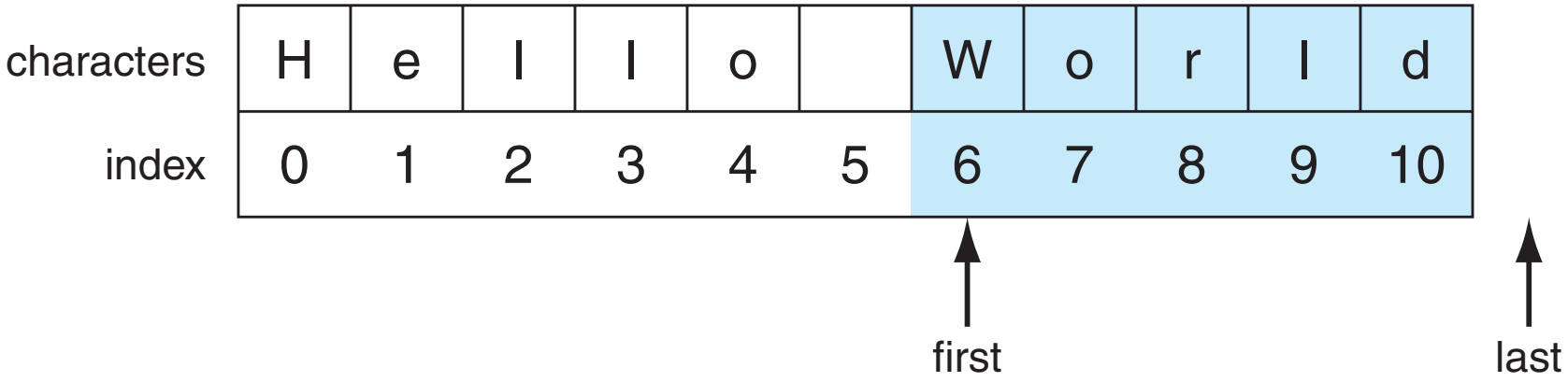



**FIGURE 4.2** Indexing subsequences with slicing.

`helloString[6:]`




`helloString[:5]`

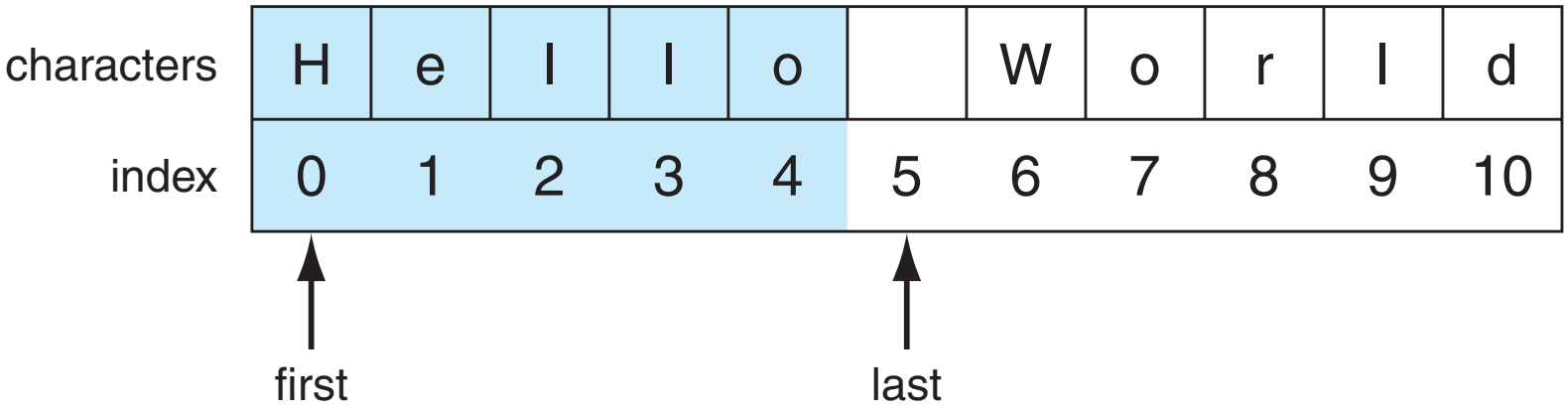



**FIGURE 4.3** Two default slice examples.

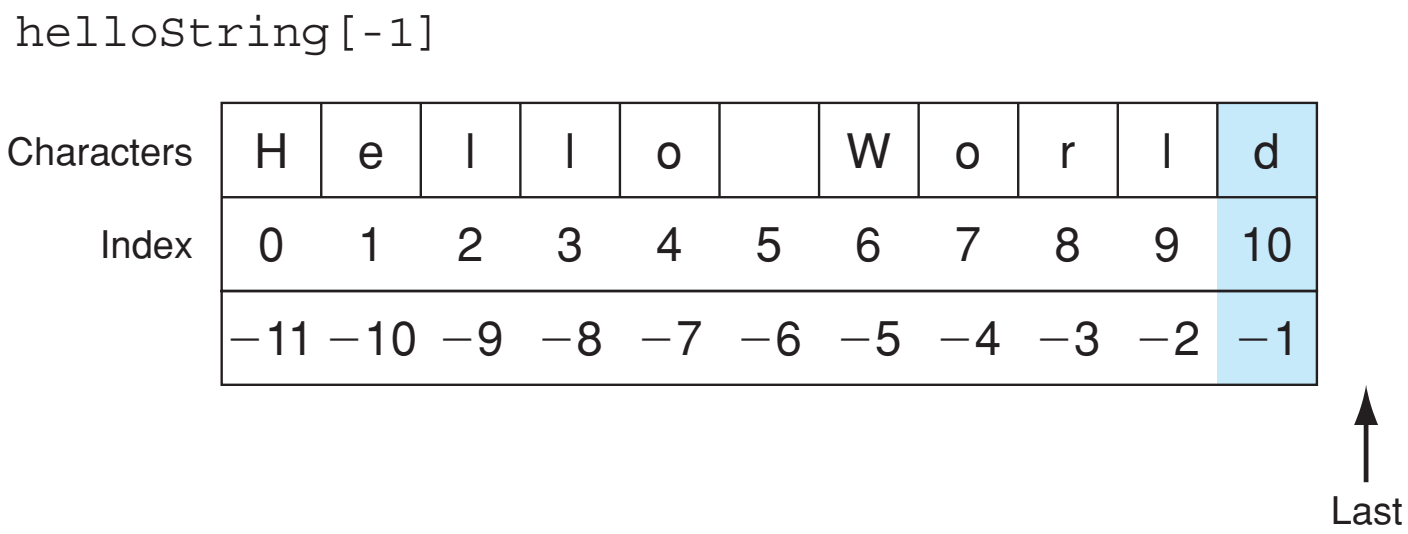



**FIGURE 4.4** Negative indices.

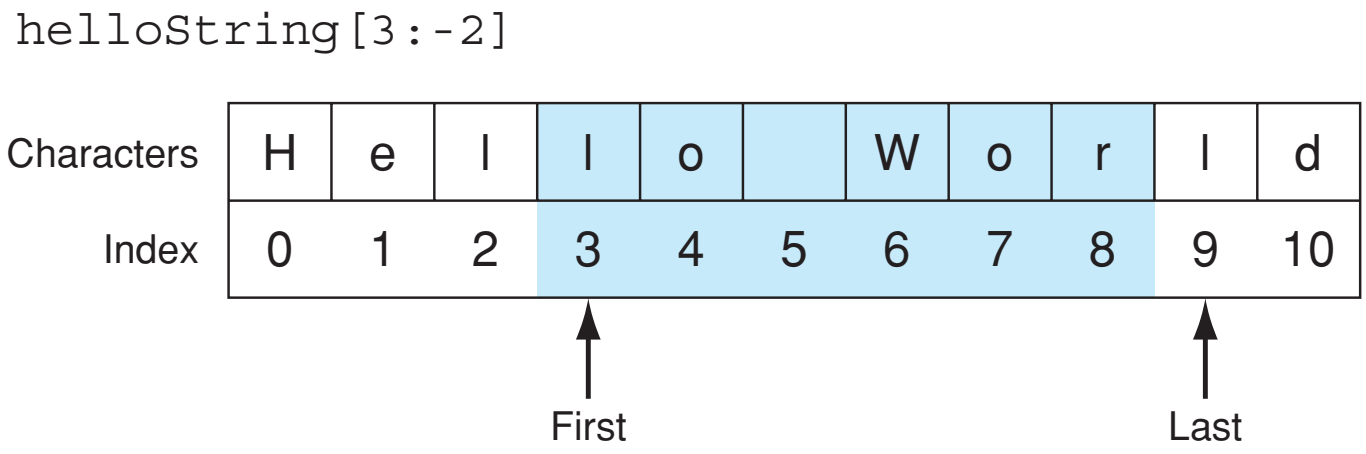



**FIGURE 4.5** Another slice example.

# Extended Slicing

- also takes three arguments:
  - `[start:finish:countBy]`
- defaults are:
  - `start` is beginning, `finish` is end, `countBy` is 1

```
my_str = 'hello world'
my_str[0:11:2] ⇒ 'hlowrd'
```

- every other letter

```
helloString[::2]
```

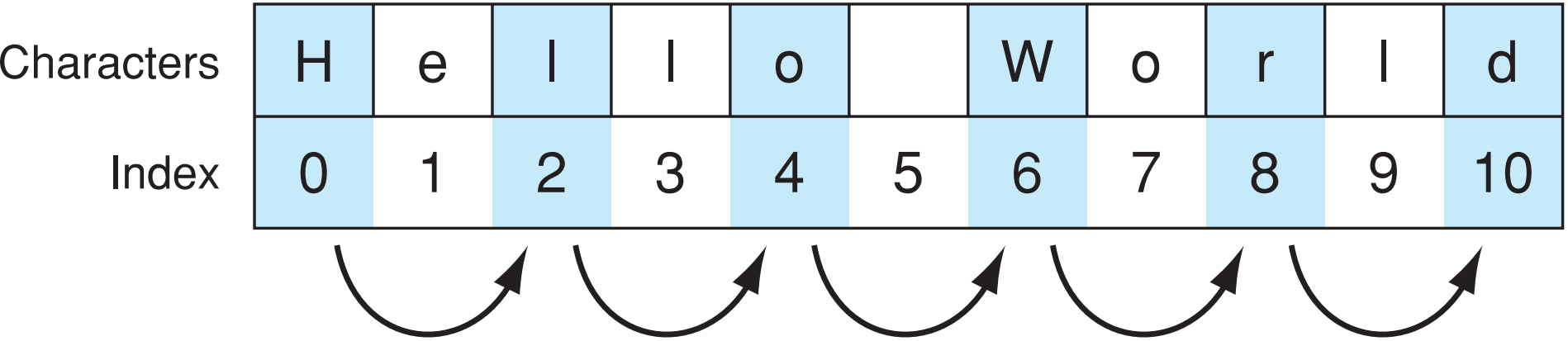



**FIGURE 4.6** Slicing with a step.

# Some Python Idioms

- idioms are python “phrases” that are used for a common task that might be less obvious to non-python folk

- how to make a copy of a string:

```
my_str = 'hi mom'
new_str = my_str[:]
```

- how to reverse a string

```
my_str = "madam I'm adam"
reverseStr = my_str[::-1]
```

# String Operations

# Sequences are Iterable

The for loop iterates through each element of a sequence in order. For a string, this means character by character:

```
>>> for char in 'Hi mom':
        print(char, type(char))


H <class 'str'>
i <class 'str'>
  <class 'str'>
m <class 'str'>
o <class 'str'>
m <class 'str'>
>>>
```

# Basic String Operations

```
s = 'spam'
```

- length operator len()

```
len(s) ⇒ 4
```

- + is concatenate

```
new_str = 'spam' + '-' + 'spam-'
print(new_str) ⇒ spam-spam-
```

- * is repeat, the number is how many times

```
new_str * 3 ⇒
'spam-spam-spam-spam-spam-spam-'
```

# Some Details on String Operations

- both + and * on strings makes a new string, does not modify the arguments
- order of operation is important for concatenation, irrelevant for repetition
- the types required are specific. For concatenation you need two strings, for repetition a string and an integer

# What does a + b mean?

- what operation does the above represent? It depends on the types!
  - two strings, concatenation
  - two integers addition
- the operator + is ***overloaded***.
  - The operation + performs depends on the types it is working on

# The `type` Function

- You can check the type of the value associated with a variable using `type`

```
my_str = 'hello world'
type(my_str) ⇒  <type 'str'>
my_str = 245
type(my_str) ⇒ <type 'int'>
```

# String Comparisons, Single Char

- Python 3 uses the Unicode mapping for characters.
  - Allows for representing non-English characters
- UTF-8, subset of Unicode, takes the English letters, numbers and punctuation marks and maps them to an integer.
- Single character comparisons are based on that number

# Comparisons Within Sequence

- It makes sense to compare within a sequence (lower case, upper case, digits).
  - `'a' < 'b'` → `True`
  - `'A' < 'B'` → `True`
  - `'1' < '9'` → `True`
- Can be weird outside of the sequence
  - `'a' < 'A'` → `False`
  - `'a' < '0'` → `False`

# Whole Strings

- Compare the first element of each string
  - if they are equal, move on to the next character in each
  - if they are not equal, the relationship between those to characters are the relationship between the string
  - if one ends up being shorter (but equal), the shorter is smaller

# Examples

- `'a' < 'b'` → `True`
- `'aaab' < 'aaac'`
  - first difference is at the last char. `'b'<'c'` so `'aaab'` is less than `'aaac'`.`True`
- `'aa' < 'aaz'`
  - The first string is the same but shorter. Thus it is smaller. `True`

# Membership Operations

- can check to see if a substring exists in the string, the **`in`** operator. Returns True or False

```
my_str = 'aabbccdd'
'a' in my_str ⇒ True
'abb' in my_str ⇒ True
'x' in my_str ⇒ False
```

# Strings are Immutable

- Strings are immutable, that is you cannot change one once you make it:

```
a_str = 'spam'
a_str[1] = 'l' → ERROR
```

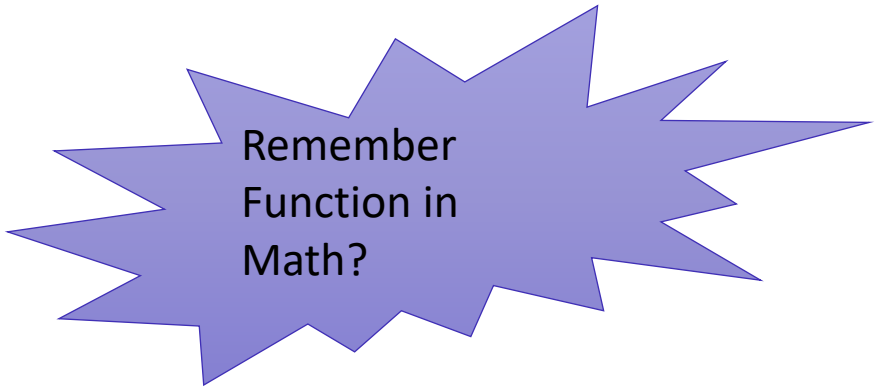
- However, you can use it to make another string (copy it, slice it, etc.)

```
new_str = a_str[:1] + 'l' + a_str[2:]
a_str →  'spam'
new_str →'slam'
```

# String Methods and Functions

# Functions, First Cut

- A function is a program that performs some operation. Its details are hidden (encapsulated), only it's interface provided.
- A function takes some number of inputs (arguments) and returns a value based on the arguments and the function's operation



# String function: `len`

- The `len` function takes as an argument a string and returns an integer, the length of a string.

```
my_str = 'Hello World'
len(my_str) ⇒ 11  # space counts!
```

# String Method

- a ***method*** is a variation on a function
  - like a function, it represents a program
  - like a function, it has input arguments and an output
- Unlike a function, it is applied in the context of a particular object.
- This is indicated by the *dot notation* invocation

# Example

- **`upper`** is the name of a method. It generates a new string that has all upper case characters of the string it was called with.

```
my_str = 'Python Rules!'
my_str.upper()  ⇒ 'PYTHON RULES!'
```

- The `upper()` method was called in the context of `my_str`, indicated by the dot between them.

# More dot notation

- in general, dot notation looks like:
  - `object.method(…)`
- It means that the object in front of the dot is calling a method that is associated with that object's type.
- The method's that can be called are tied to the type of the object calling it. Each type has different methods.

# Find

```
my_str = 'hello'
my_str.find('l')          # find index of 'l' in my_str
```

⇒ 2

Note how the method 'find' operates on the string object my_str and the two are associated by using the “dot” notation: my_str.find('l').

Terminology: the thing(s) in parenthesis, i.e. the 'l' in this case, is called an **argument**.

# Chaining methods

Methods can be chained together.

- Perform first operation, yielding an object
- Use the yielded object for the next method

```
my_str = 'Python Rules!'
my_str.upper() ⇒  'PYTHON RULES!'
my_str.upper().find('O')
⇒ 4
```

# Optional Arguments

Some methods have optional arguments:

- if the user doesn't provide one of these, a default is assumed
- find has a default second argument of 0, where the search begins

```
a_str = 'He had the bat'
a_str.find('t') ⇒ 7 # 1st 't',start at 0
a_str.find('t',8) ⇒ 13 # 2nd 't'
```

# Nesting Methods

- You can “nest” methods, that is the result of one method as an argument to another
- remember that parenthetical expressions are done “inside out”: do the inner parenthetical expression first, then the next, using the result as an argument

```
a_str.find('t', a_str.find('t')+1)
```

- translation: find the second 't'.

# How to know?

- How can you determine the methods associated with a type, and once you find the name,
- how can you determine the arguments?
- You can ask iPython to show you all the potential methods for an object.
- Your iPython will show all the methods available for an object of that type if you type the object (or a variable of that type), the dot (.), and then a tab character.
- Your iPython will respond by providing a list of all the potential methods that can be invoked with an object of that type

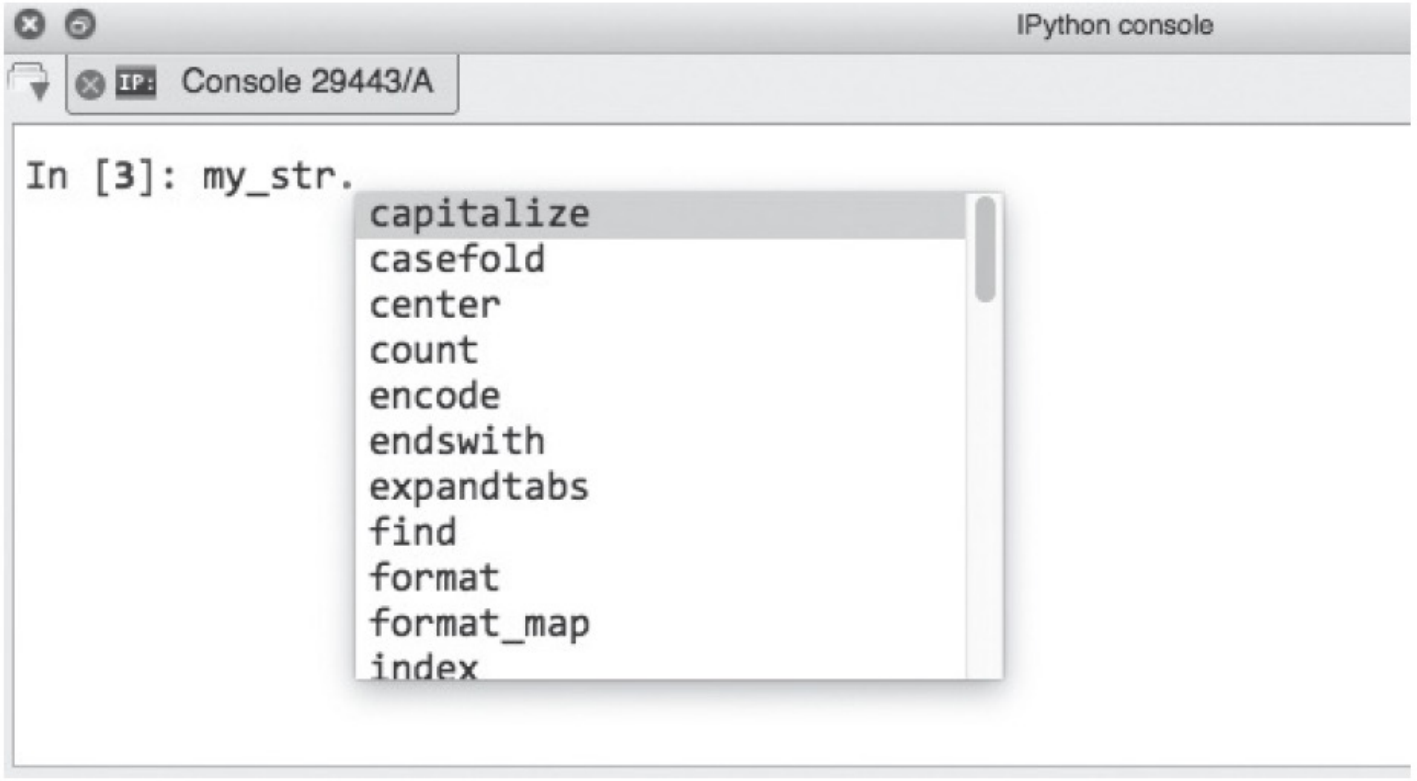



**FIGURE 4.7** In your IDE, tab lists potential methods. [Screenshot from Python. Copyright © by Python. Used by permission of Python.]


**FIGURE 4.8** In iPython, tab lists potential methods, with leading letter. [Screenshot from Python. Copyright © by Python. Used by permission of Python.]

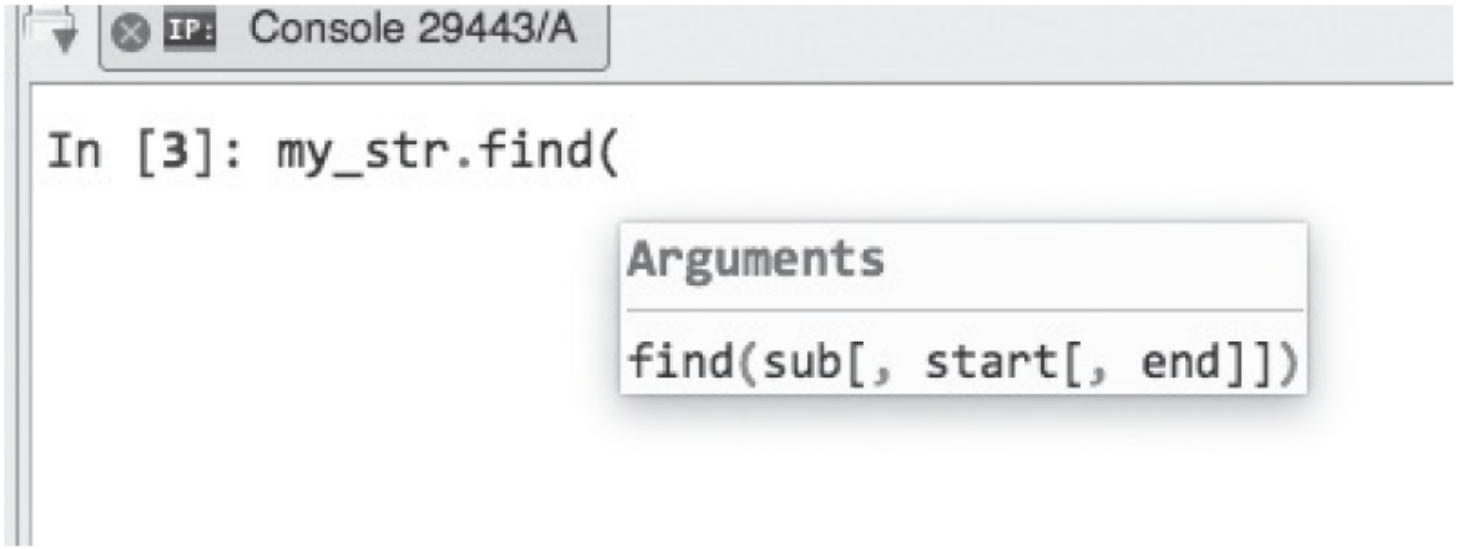



**FIGURE 4.9** Your IDE pop-up provides help with function arguments. [Screenshot from Python. Copyright © by Python. Used by permission of Python.]

| | |
|---|---|
| `capitalize( )` | `lstrip( [`*chars*`])` |
| `center(` *width*`[,` *fillchar*`])` | `partition(` *sep*`)` |
| `count(` *sub*`[,` *start*`[,` *end*`]])` | `replace(` *old*`,` *new*`[,` *count*`])` |
| `decode( [`*encoding*`[,` *errors*`]])` | `rfind(` *sub* `[,`*start*`[,`*end*`]])` |
| `encode( [`*encoding*`[,`*errors*`]])` | `rindex(` *sub*`[,` *start*`[,` *end*`]])` |
| `endswith(` *suffix*`[,` *start*`[,` *end*`]])` | `rjust(` *width*`[,` *fillchar*`])` |
| `expandtabs( [`*tabsize*`])` | `rpartition(`*sep*`)` |
| `find(` *sub*`[,` *start*`[,` *end*`]])` | `rsplit( [`*sep* `[,`*maxsplit*`]])` |
| `index(` *sub*`[,` *start*`[,` *end*`]])` | `rstrip( [`*chars*`])` |
| `isalnum( )` | `split( [`*sep* `[,`*maxsplit*`]])` |
| `isalpha( )` | `splitlines( [`*keepends*`])` |
| `isdigit( )` | `startswith(` *prefix*`[,` *start*`[,` *end*`]])` |
| `islower( )` | `strip( [`*chars*`])` |
| `isspace( )` | `swapcase( )` |
| `istitle( )` | `title( )` |
| `isupper( )` | `translate(` *table*`[,` *deletechars*`])` |
| `join(`*seq*`)` | `upper( )` |
| `lower( )` | `zfill(` *width*`)` |
| `ljust( width[,` *fillchar*`])` | |

**TABLE 4.2** Python String Methods

# Formatted Output for String

# String Formatting, Better Printing

- So far, we have just used the defaults of the print function
- We can do many more complicated things to make that output “prettier” and more pleasing.
- We will use it in our display function

# Basic Form

- To understand string formatting, it is probably better to start with an example.

```
print("Sorry, is this the {} minute
  {}?".format(5, 'ARGUMENT'))
```

prints `Sorry, is this the 5 minute ARGUMENT`

# format method

- `format` is a method that creates a new string where certain elements of the string are re-organized i.e., *formatted*
- The elements to be re-organized are the curly bracket elements in the string.
- Formatting is complicated, this is just some of the easy stuff (see the docs)

# map args to { }

- The string is modified so that the { } elements in the string are replaced by the format method arguments
- The replacement is in order: first { } is replaced by the first argument, second { } by the second argument and so forth.

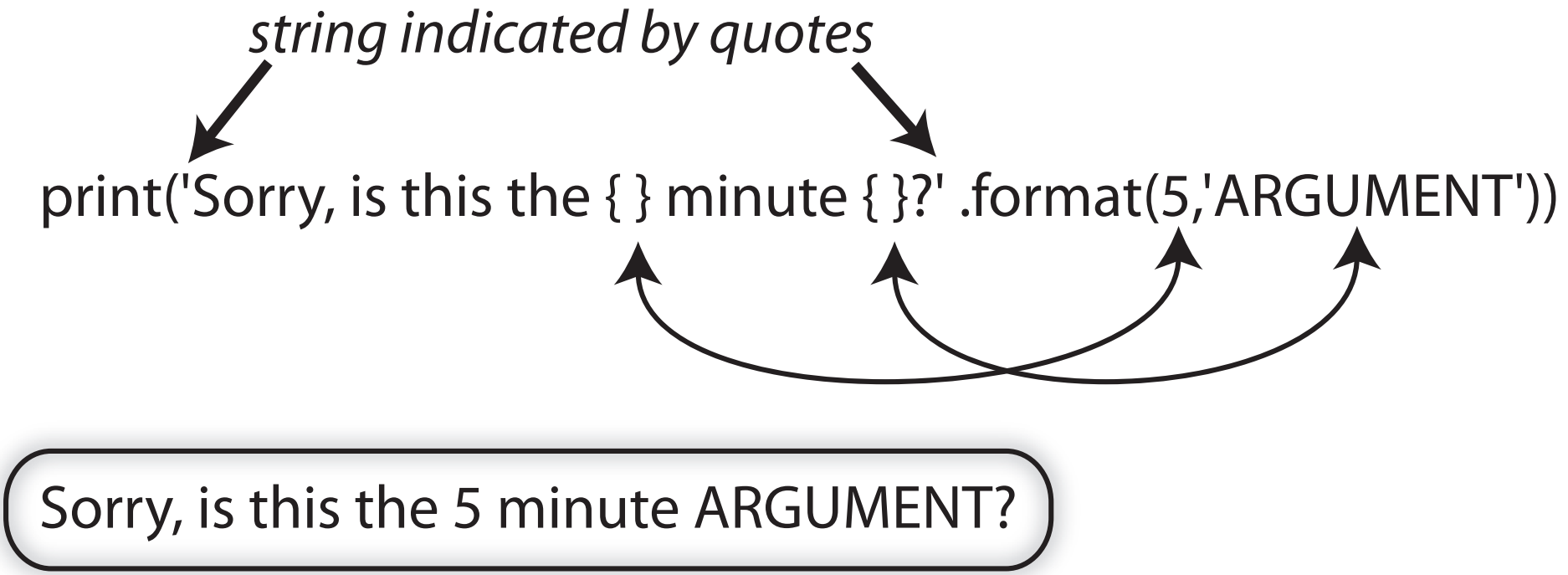



**FIGURE 4.10** String formatting example.

# Format String

- the content of the curly bracket elements are the format string, descriptors of how to organize that particular substitution.
  - types are the kind of thing to substitute, numbers indicate total spaces.

| | |
|---|---|
| s | string |
| d | decimal integer |
| f | floating-point decimal |
| e | floating-point exponential |
| % | floating-point as percent |

**TABLE 4.3** Most commonly used types.

| | |
|---|---|
| < | left |
| > | right |
| ^ | center |

**TABLE 4.4** Width alignments.

# Each format string

- Each bracket looks like

  `{:align width .precision descriptor}`

  - `align` is optional (default left for strings, right for numbers)
  - `width` is how many spaces (default just enough)
  - `.precision` is for floating point rounding (default no rounding)
  - `descriptor` is the expected type (error if the arg is the wrong type)

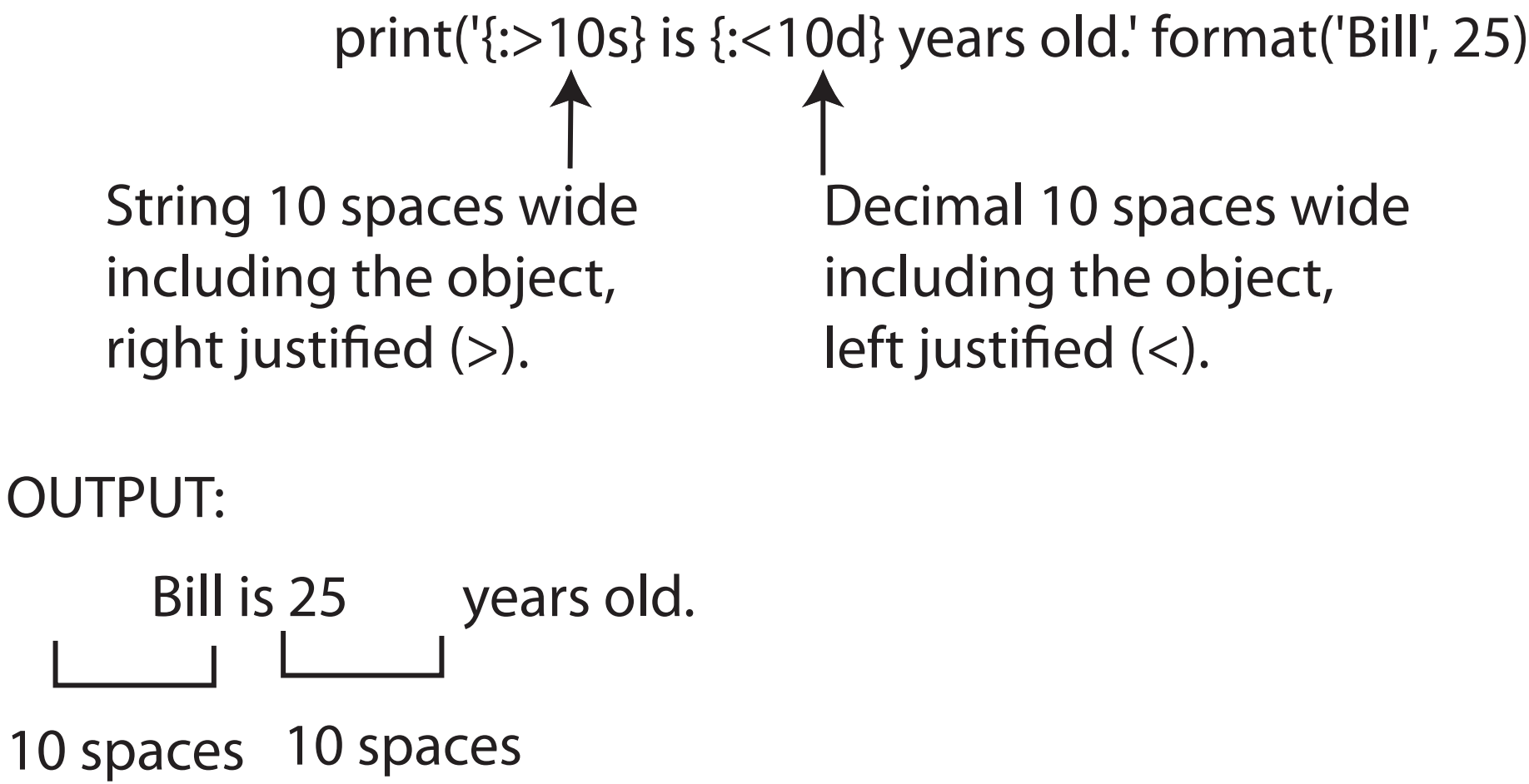



**FIGURE 4.11** String formatting with width descriptors and alignment.

# Nice table

```
>>> for i in range(5):
        print("{:10d} --> {:4d}".format(i,i**2))

         0 -->    0
         1 -->    1
         2 -->    4
         3 -->    9
         4 -->   16
```

# Floating Point Precision

Can round floating point to specific number of decimal places

```
In  [1]: import math
In  [2]: print(math.pi) # unformatted printing
3.141592653589793
In  [3]: print("Pi is {:.4f}".format(math.pi)) # floating-point precision 4
Pi is 3.1416
In  [4]: print("Pi is {:8.4f}".format(math.pi)) # specify both precision and width
Pi is   3.1416
In  [5]: print("Pi is {:8.2f}".format(math.pi))
Pi is     3.14
```

# C-style formatting

- Uses % sign rather than {} - easier and more commonly used
- print (“%d feet and %d inches is %1.4f meters” % (feet, inches, meters) )
- if feet = 5 and inches = 2, prints
- 5 feet and 2 inches is 1.5748 meters
- same specifiers - %d integer, %f float and %s string

# C-style Formatting

- Width modifiers come before field specifiers
- print (“%6d feet and %6d inches” % (feet, inches) )
- prints (- is a space)
- -----5 feet and -----2 inches
- standard is right justified. If you want left, make the width modifier negative

# More String Formatting

# Control and String

# Iteration Through a Sequence

- To date we have seen the while loop as a way to iterate over a suite (a group of python statements)
- We briefly touched on the for statement for iteration, such as the elements of a list or a string
- We use the for statement to process each element of a list, one element at a time

```
for item in sequence:
    suite
```

# What `for` means

```
my_str='abc'
for char in 'abc':
    print(char)
```

- first time through, char = 'a' (my_str[0])
- second time through, char='b' (my_str[1])
- third time through, char='c' (my_str[2])
- no more sequence left, for ends

# Power of the for statement

- Sequence iteration as provided by the for state is very powerful and very useful in python.
- Allows you to write some very “short” programs that do powerful things.

# Code Listing 4.1: Find a Letter

```
# Our implementation of the find function. Prints the index where
# the target is found; a failure message, if it isn't found.
# This version only searches for a single character.

river = 'Mississippi'
target = input('Input a character to find: ')
for index in range(len(river)):                 # for each index
    if river[index] == target:                  # check if the target is found
        print("Letter found at index: ", index)  # if so, print the index
        break                                   # stop searching
else:
    print('Letter',target,'not found in',river)
```

# Code Listing 4.2: Find with enumerate

## Enumerate function

- The enumerate function prints out two values: the index of an element and the element itself
- Can use it to iterate through both the index and element simultaneously, doing dual assignment

```
# Our implementation of the find function. Prints the index where
# the target is found; a failure message, if it isn't found.
# This version only searches for a single character.

river = 'Mississippi'
target = input('Input a character to find: ')
for index,letter in enumerate(river):                # for each index
    if letter == target:                             # check if the target is found
        print("Letter found at index: ", index)      # if so, print the index
        break                                        # stop searching
else:
    print('Letter',target,'not found in',river)
```

# Working with String

# split function

- The `split` function will take a string and break it into multiple new string parts depending on the argument character.
- by default, if no argument is provided, split is on any whitespace character (tab, blank, etc.)
- you can assign the pieces with multiple assignment if you know how many pieces are yielded.

# reorder a name

```
In [1]: name = 'John Marwood Cleese'
In [2]: first, middle, last = name.split()
In [3]: transformed = last + ', ' + first
+ ' ' + middle
In [4]: print(transformed)
Cleese, John Marwood
In [5]: print(name)
John Marwood Cleese
In [6]: print(first)
John
In [7]: print(middle)
Marwood
```

# reorder a name

```
In [1]: name = 'John Marwood Cleese'
In [8]: print(last)
Cleese

In [9]: first, middle = name.split() # e r r o r : not enough p i e c e s
Traceback (most recent call last):
File "<pyshell#71>", line 1, in <module>
first, middle = name.split()
ValueError: too many values to unpack

In [10]: first, middle, last = name.split(' ') # s p l i t on s p a c e ' '

In [11]: print(first, middle, last)
John Marwood Cleese
```

# Visit Palindromes from previous Practice

- A palindrome is a string that prints the same forward and backward

Let’s make some code for this…

# Palindromes (cont.)

Suppose we include several new criteria to determine whether String is palindrome :

- case does not matter
- punctuation is ignored

"Madam I'm Adam" is thus a palindrome

Let’s make some code for this…

# Lower Case and Punctuation

- every letter is converted using the `lower` method
- `import string`, brings in a series of predefined sequences (`string.digits`, `string.punctuation`, `string.whitespace`)
- we remove all non-wanted characters with the `replace` method. First arg is what to replace, the second the replacement.

# Code Listing 4.4: Palindrome

```
# Palindrome tester
import string

original_str = input('Input a string:')
modified_str = original_str.lower()

bad_chars = string.whitespace + string.punctuation

for char in modified_str:
    if char in bad_chars:  # remove bad characters
        modified_str = modified_str.replace(char,'')

if modified_str == modified_str[::-1]: # it is a palindrome
    print(\
'The original string is:  {}\n\
 the modified string is: {}\n\
 the reversal is:        {}\n\
 String is a palindrome'.format(original_str, modified_str, modified_str[::-1
]))
else:
    print(\
'The original string is:  {}\n\
 the modified string is: {}\n\
 the reversal is:        {}\n\
 String is not a palindrome'.format(original_str,modified_str,modified_str[::-
1]))
```

# Reminder, four rules

1. **Think** before you program!
2. A program is a **human-readable** essay on problem solving that also happens to execute on a computer.
3. The best way to improve your programming and problem solving skills is to **practice**!
4. **Test** your code, often and thoroughly.

# Reference


- Punch & Enbody, The Practice of Computing Using Python, 3rd ed., Pearson Education, Inc., 2017
- Ljubomir Perkovic, “Introduction to Computing using Python”