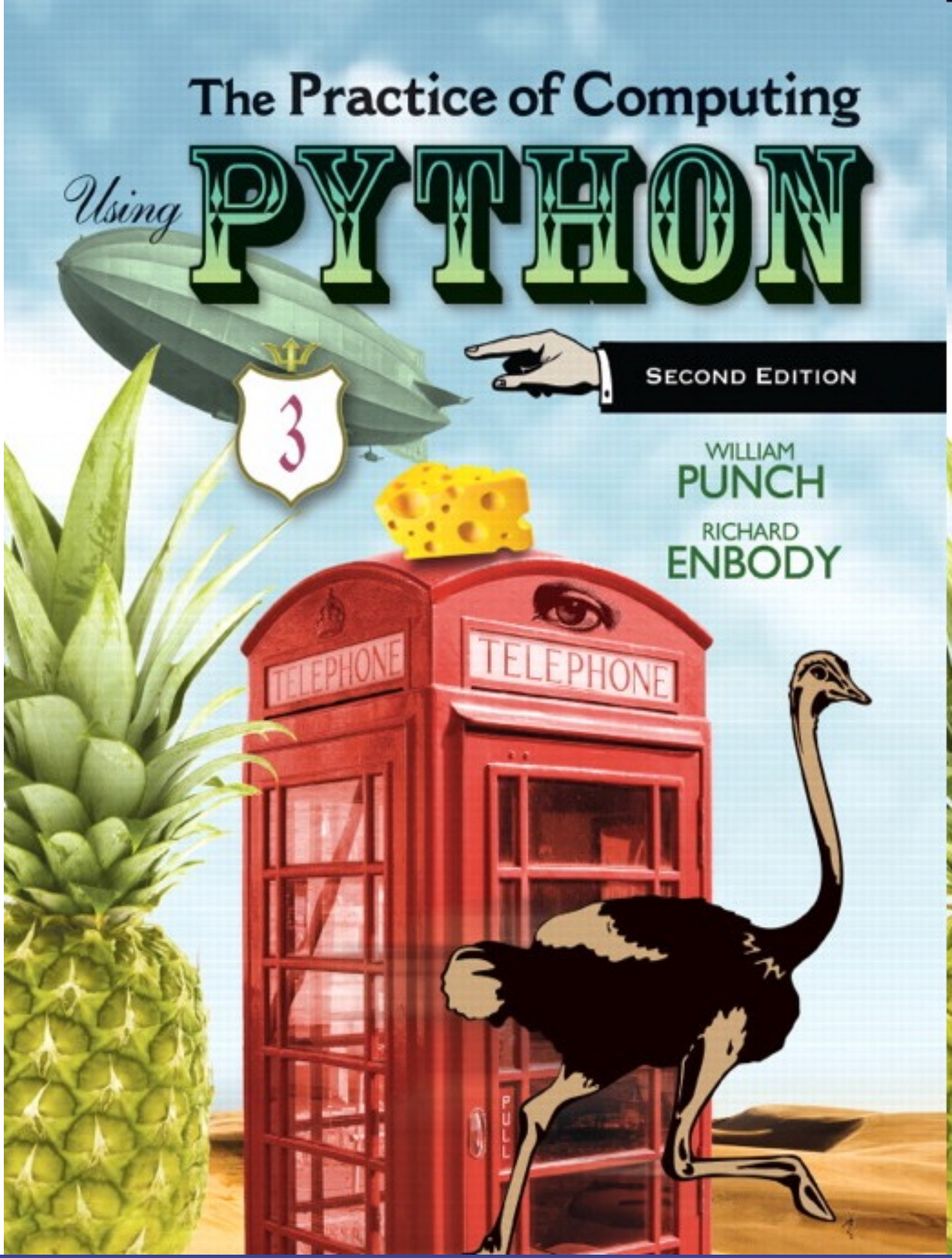
Chapter 4

Working with Strings



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

And then there is """ """

- 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"""
```

non-printing characters

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

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

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.

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 F for the full set

Strings

Can use single or double quotes:

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

Just don't mix them

- `my_str = 'hi mom"` ⇒ ERROR

Inserting an apostrophe:

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

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

characters	H	e	l	l	o		W	o	r	l	d
index	0	1	2	3	4	5	6	7	8	9	10
									...	−2	−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
```

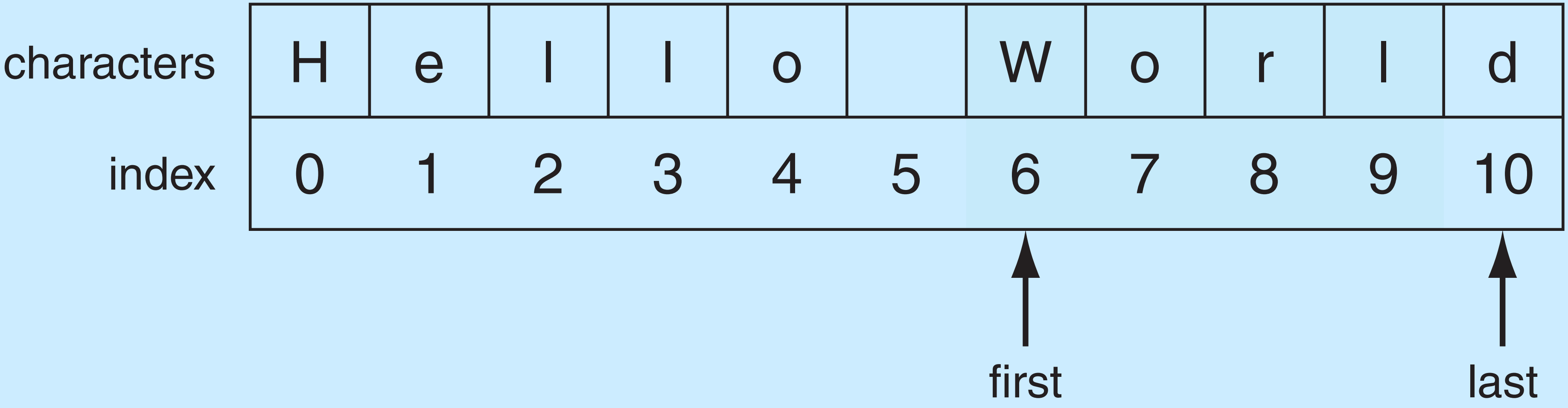
Slicing, the rules

- 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

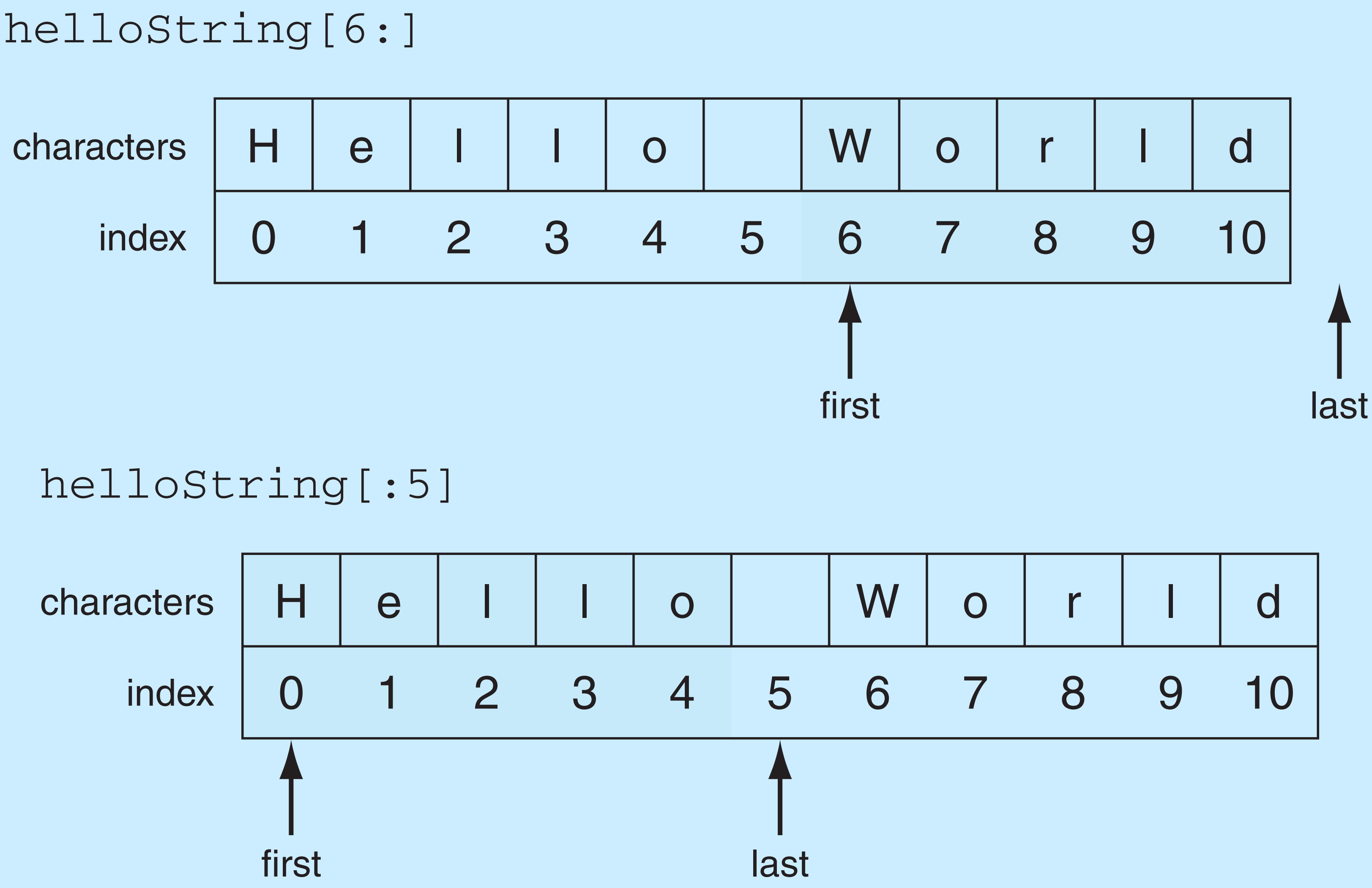
- slicing uses what is called a half-open range
- the first index is included in the sequence
- the last index is one ***after*** what is included

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



characters	H	e	l	l	o		W	o	r	l	d
index	0	1	2	3	4	5	6	7	8	9	10

FIGURE 4.2 Indexing subsequences with slicing.



```
helloString[6:]
```

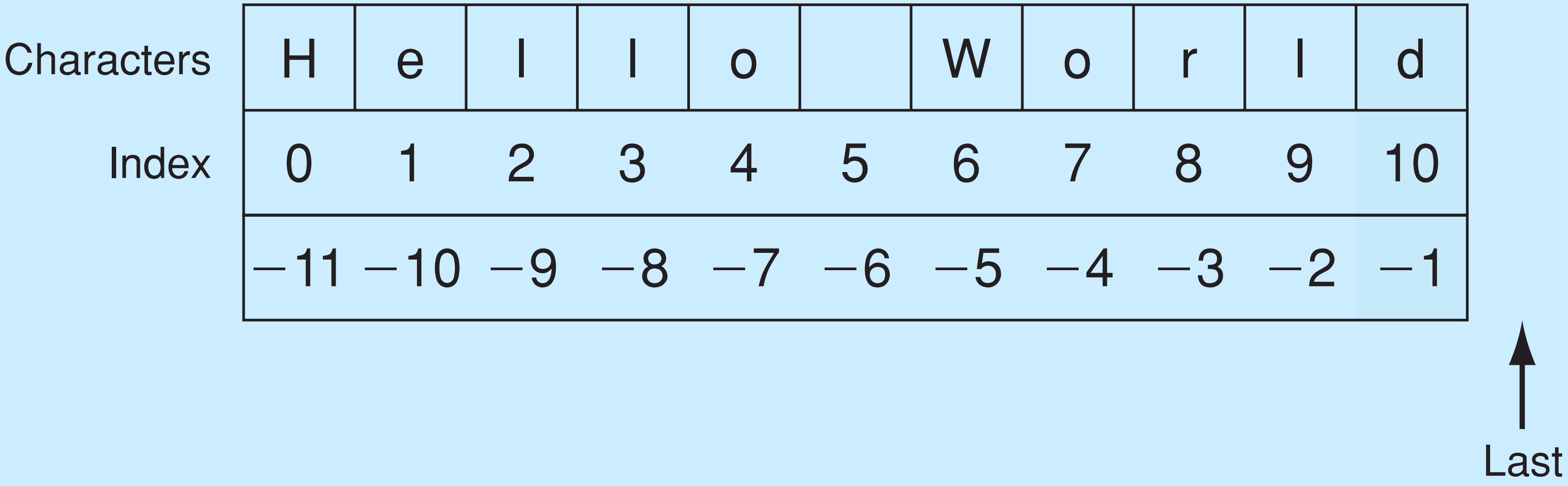
characters	H	e	l	l	o		W	o	r	l	d
index	0	1	2	3	4	5	6	7	8	9	10

```
helloString[:5]
```

characters	H	e	l	l	o		W	o	r	l	d
index	0	1	2	3	4	5	6	7	8	9	10

FIGURE 4.3 Two default slice examples.

```
helloString[-1]
```

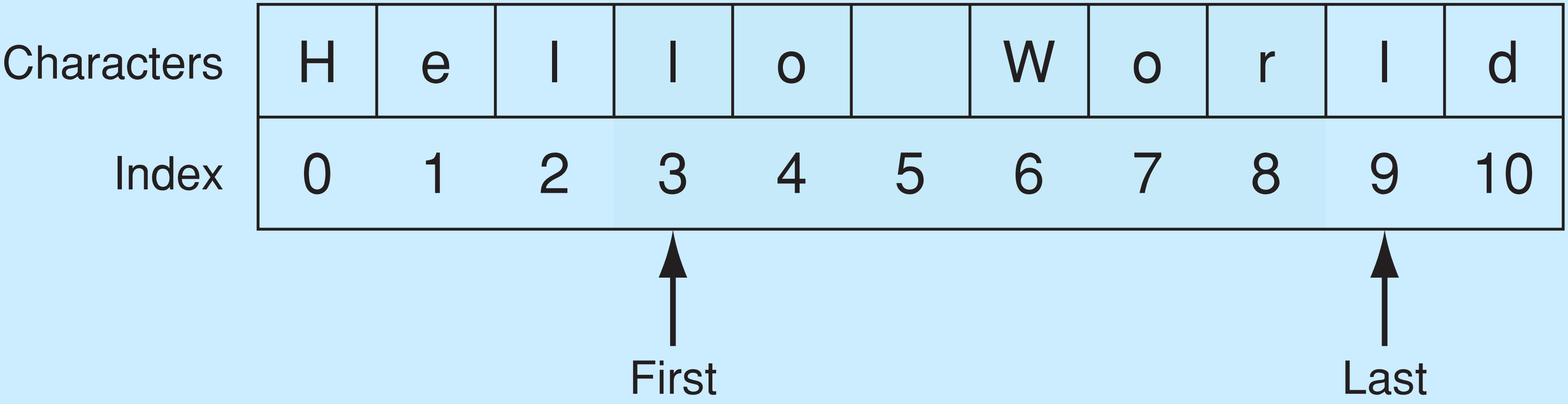


Characters	H	e	l	l	o		W	o	r	l	d
Index	0	1	2	3	4	5	6	7	8	9	10
	−11	−10	−9	−8	−7	−6	−5	−4	−3	−2	−1

Last

FIGURE 4.4 Negative indices.

```
helloString[3:-2]
```



Characters	H	e	l	l	o		W	o	r	l	d
Index	0	1	2	3	4	5	6	7	8	9	10

FIGURE 4.5 Another slice example.

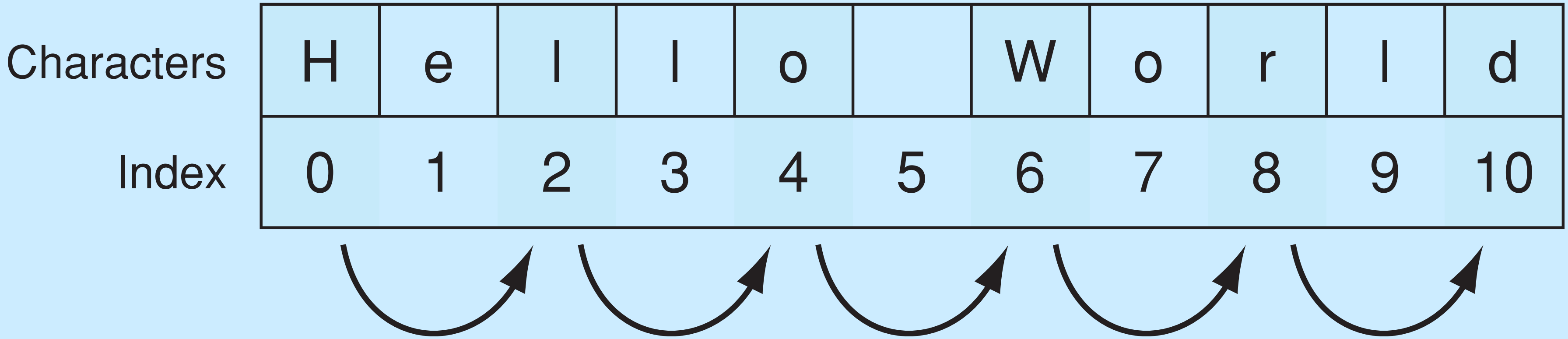
Extended Slicing

- 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]
```



Characters	H	e	l	l	o		W	o	r	l	d
Index	0	1	2	3	4	5	6	7	8	9	10

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

- 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 generation, 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 arugments:

- 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

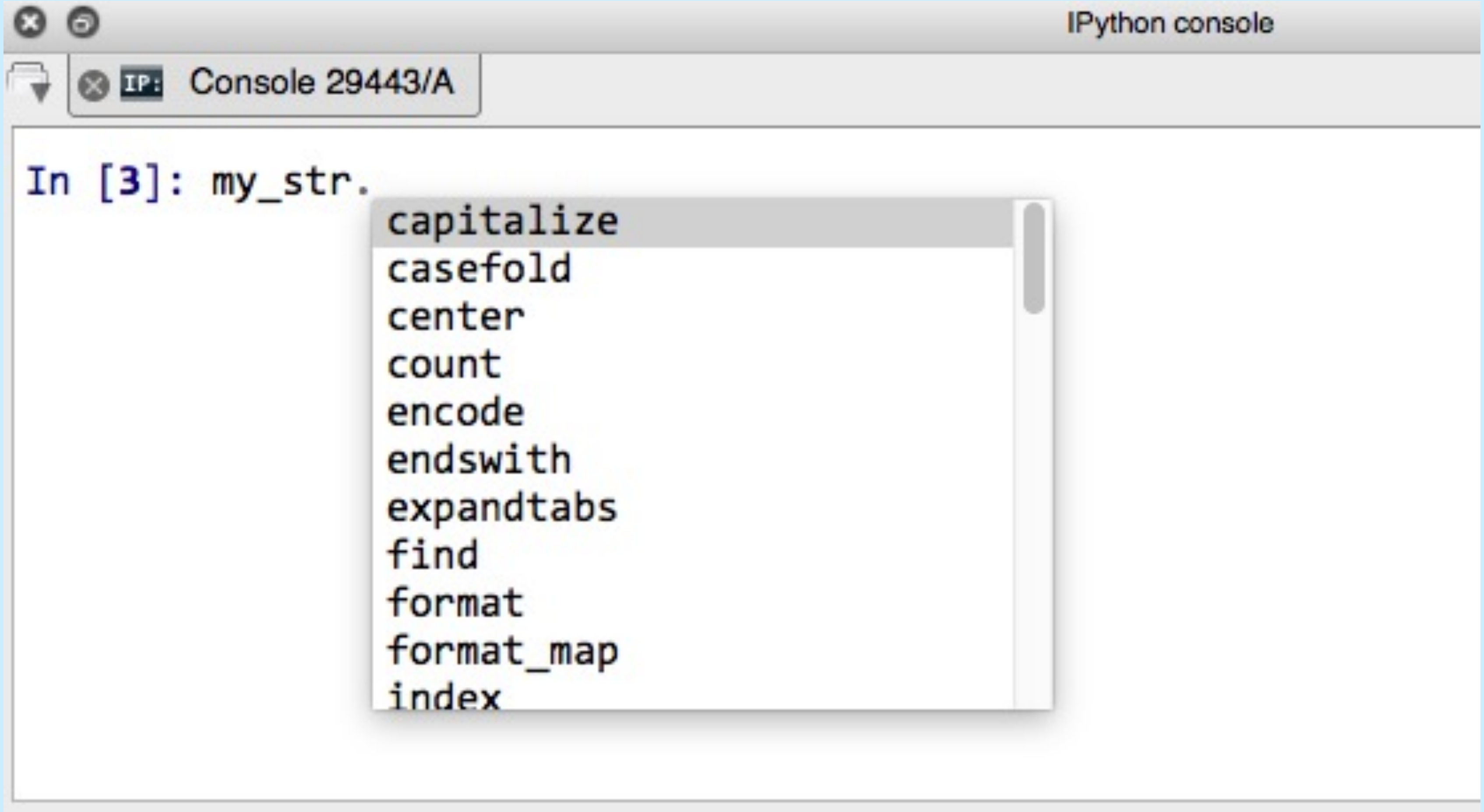
- You can “nest” methods, that is the result of one method as an argument to another
- remember that parenthetical expressions are did “inside out”: do the inner parenthetical expression first, then the next, using the result as an argument

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

- translation: find the second 't'.

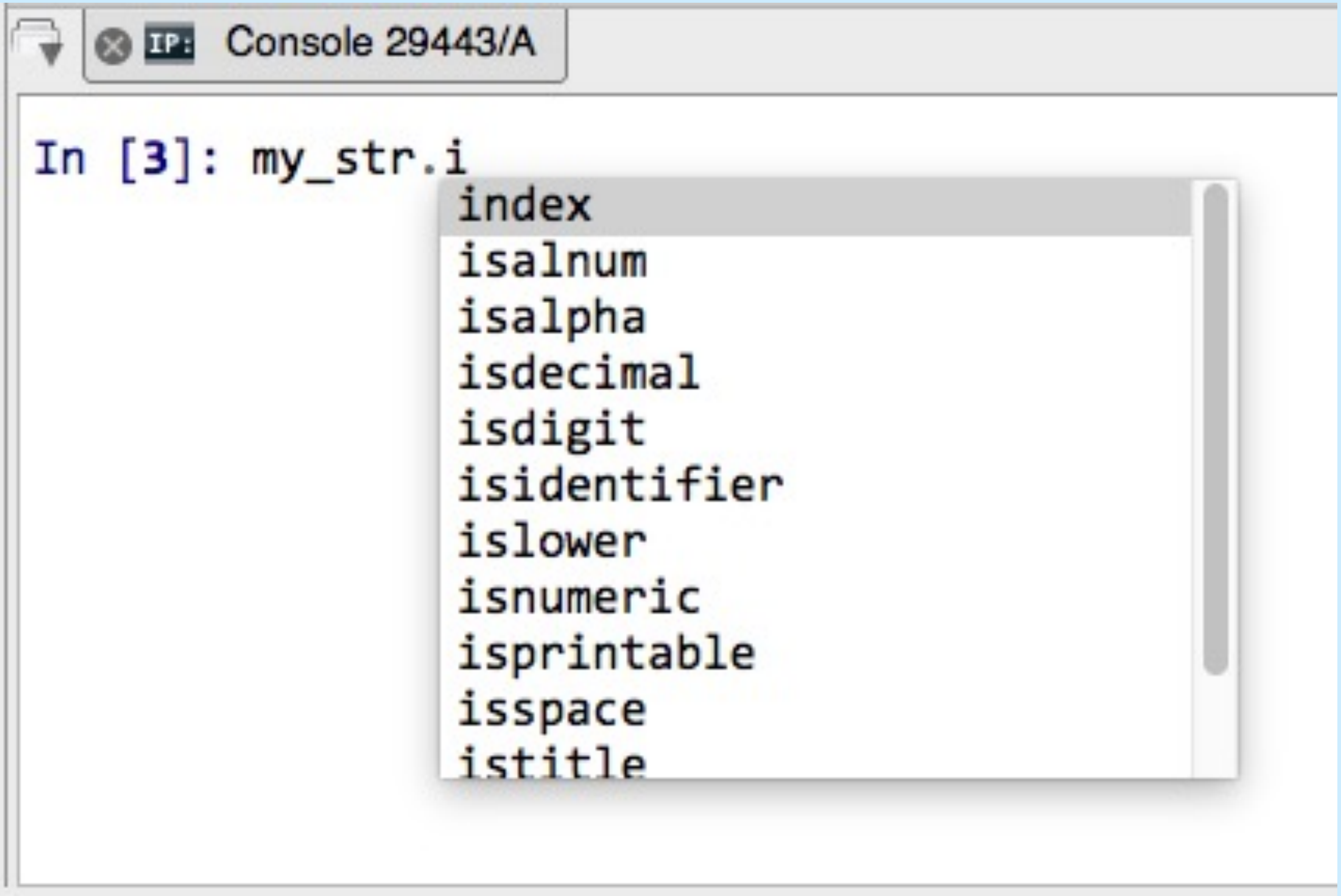
How to know?

- You can use Spyder to find available methods for any type. You enter a variable of the type, followed by the `'.'` (dot) and then a tab.
- Remember, methods match with a type. Different types have different methods
- If you type a method name, Spyder will remind you of the needed and optional arguments.



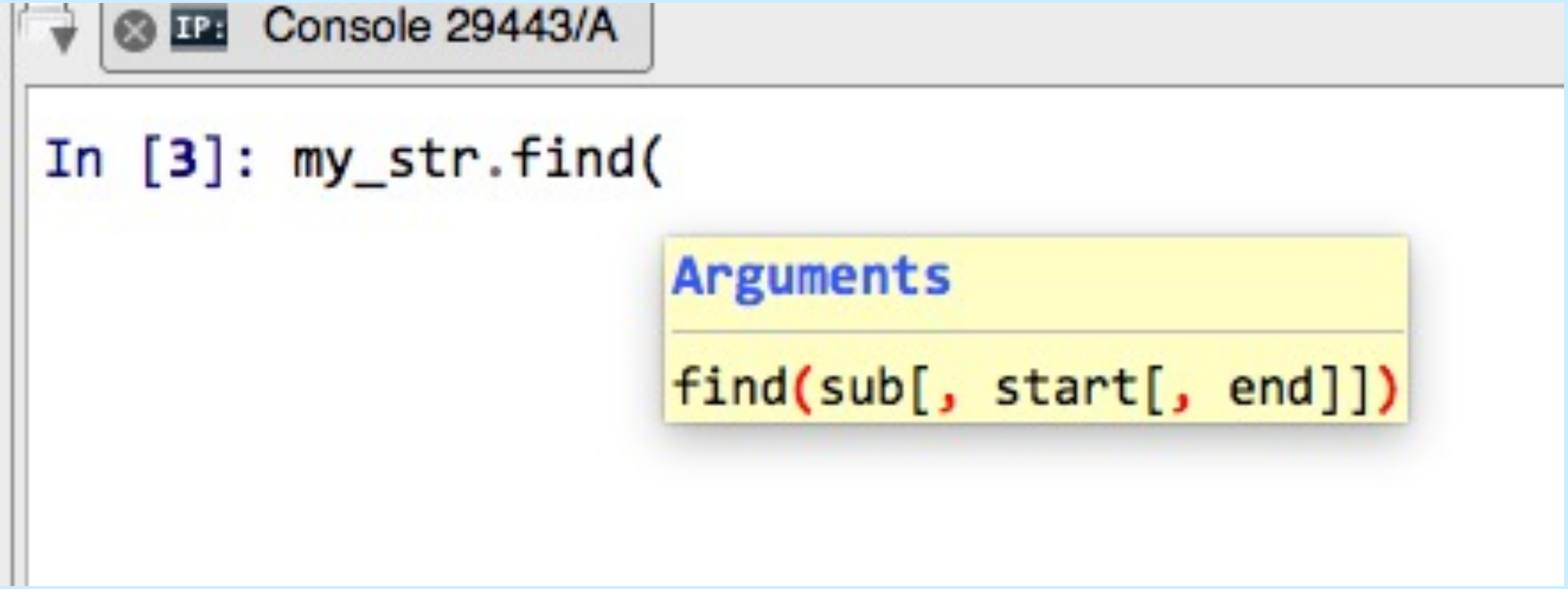
```
In [3]: my_str.
        capitalize
        casefold
        center
        count
        encode
        endswith
        expandtabs
        find
        format
        format_map
        index
```

Figure 4.7



```
In [3]: my_str.i
```

Figure 4.8



```
In [3]: my_str.find(
```

```
find(sub[, start[, end]])
```

Figure 4.9

`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

String formatting

CSE 231, Bill Punch

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'))
```

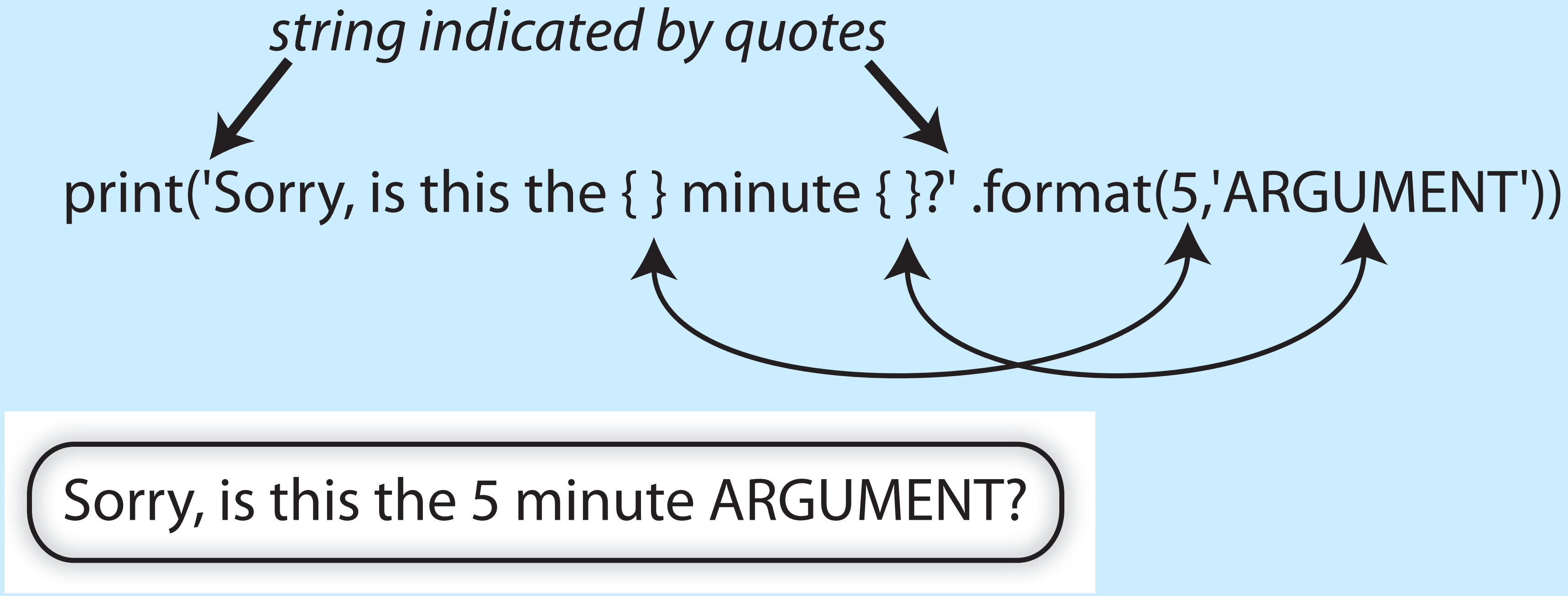
prints `Sorry, is this the 5 minute ARGUMENT`

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
- They replacement is in order: first { } is replaced by the first argument, second { } by the second argument and so forth.



string indicated by quotes

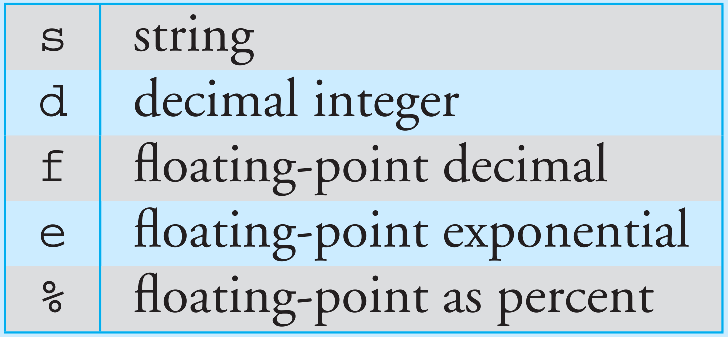
```
print('Sorry, is this the { } minute { }?' .format(5,'ARGUMENT'))
```

```
Sorry, is this the 5 minute ARGUMENT?
```

FIGURE 4.10 String formatting example.

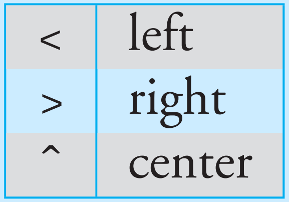
Format string

- 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

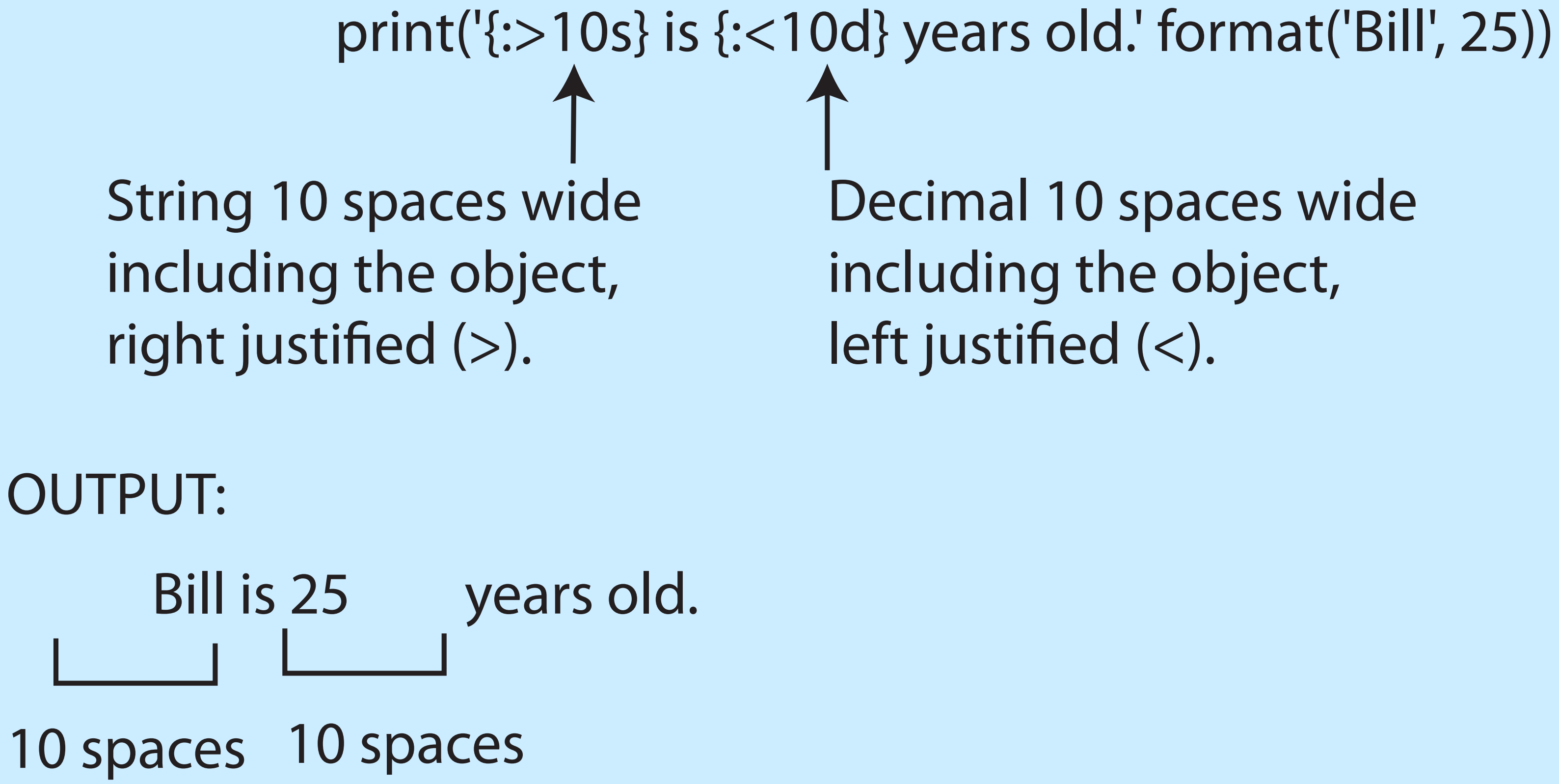
TABLE 4.4 Width alignments.

Each format string

- Each bracket looks like

```
{:align width .precision descriptor}
```

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



```
print('{:>10s} is {:<10d} years old.' format('Bill', 25))
```

String 10 spaces wide
including the object,
right justified (>).

Decimal 10 spaces wide
including the object,
left justified (<).

OUTPUT:

```
      Bill is 25         years old.
```

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

```
>>> import math
>>> print(math.pi)                        # unformatted printing
3.141592653589793
>>> print("Pi is {:.4f}".format(math.pi)) # floating—point precision 4
Pi is 3.1416
>>> print("Pi is {:8.4f}".format(math.pi)) # specify both precision and width
Pi is   3.1416
>>> print("Pi is {:8.2f}".format(math.pi))
Pi is     3.14
```

iteration

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

for statement

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

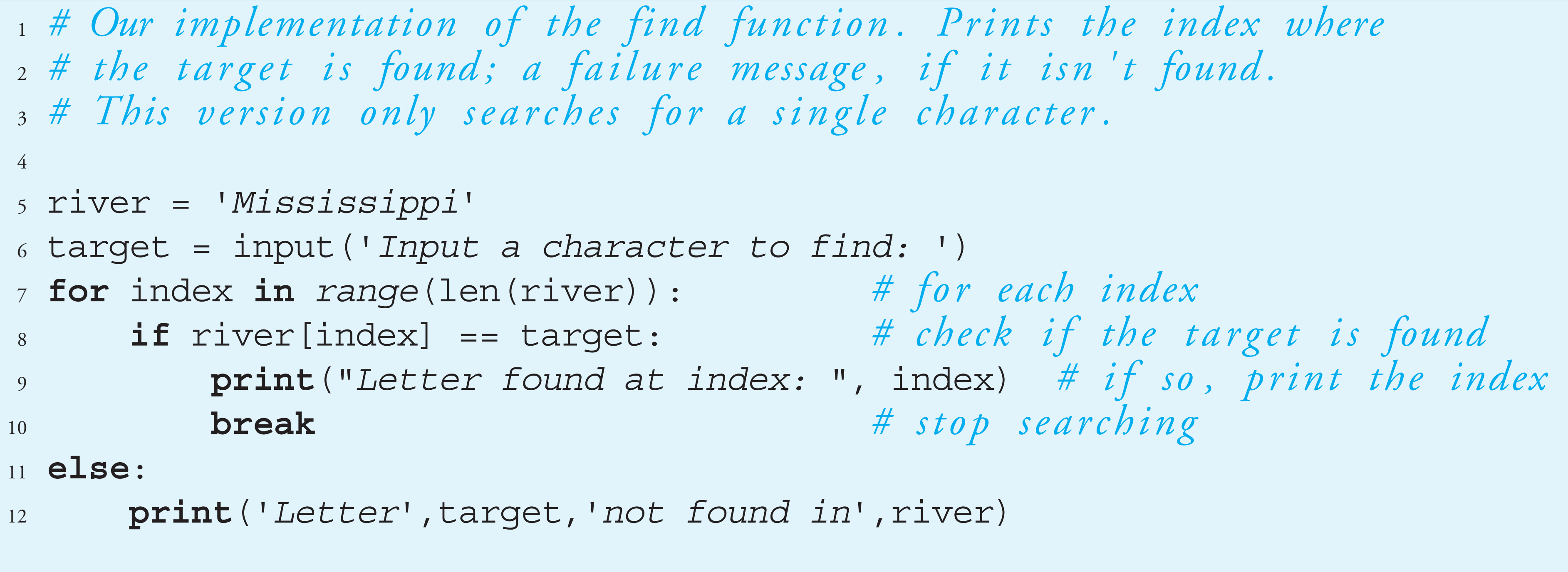
```
my_str='abc'
for char in 'abc':
    print(char)
```

- 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



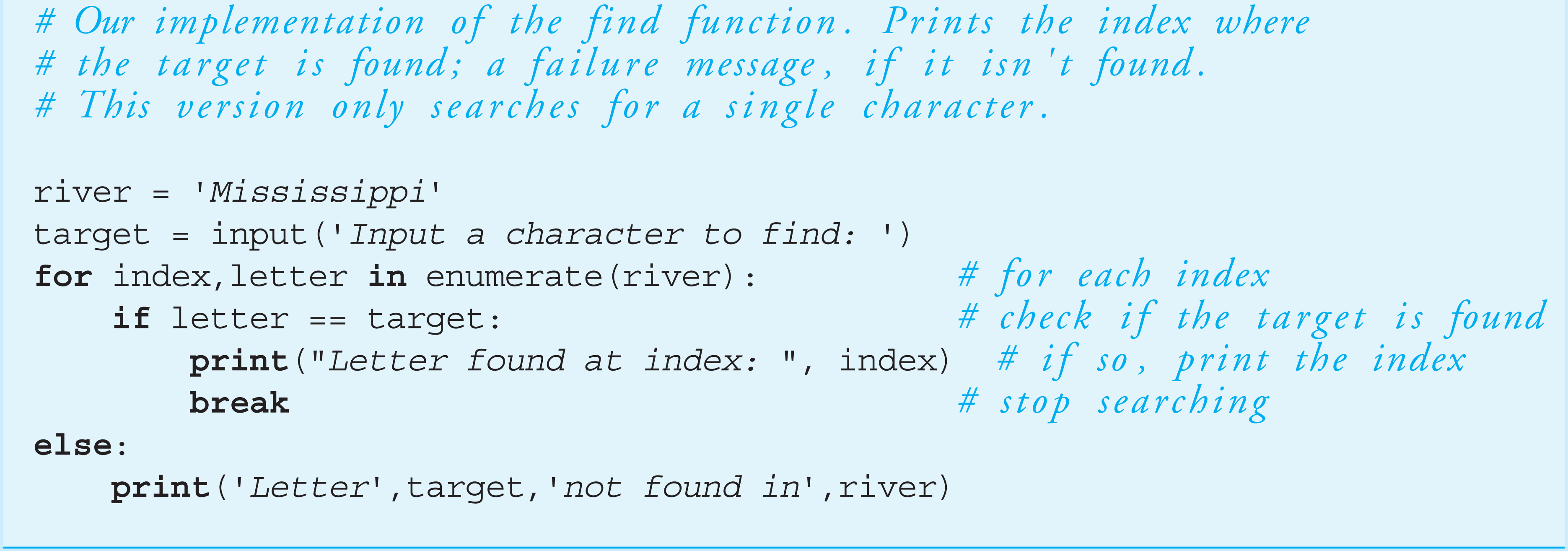
```
# Our implementation of the find function. Prints the index where
# the target is found; a failure message, if it isn't found.
# This version only searches for a single character.

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)
```

"The Practice of Computing Using Python, 3rd Edition",
Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

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

Code Listings 4.2
find with enumerate



```
# 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)
```

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

```
>>> name = 'John Marwood Cleese'
>>> first, middle, last = name.split()
>>> transformed = last + ', ' + first + ' ' + middle
>>> print(transformed)
Cleese, John Marwood
>>> print(name)
John Marwood Cleese
>>> print(first)
John
>>> print(middle)
Marwood
```

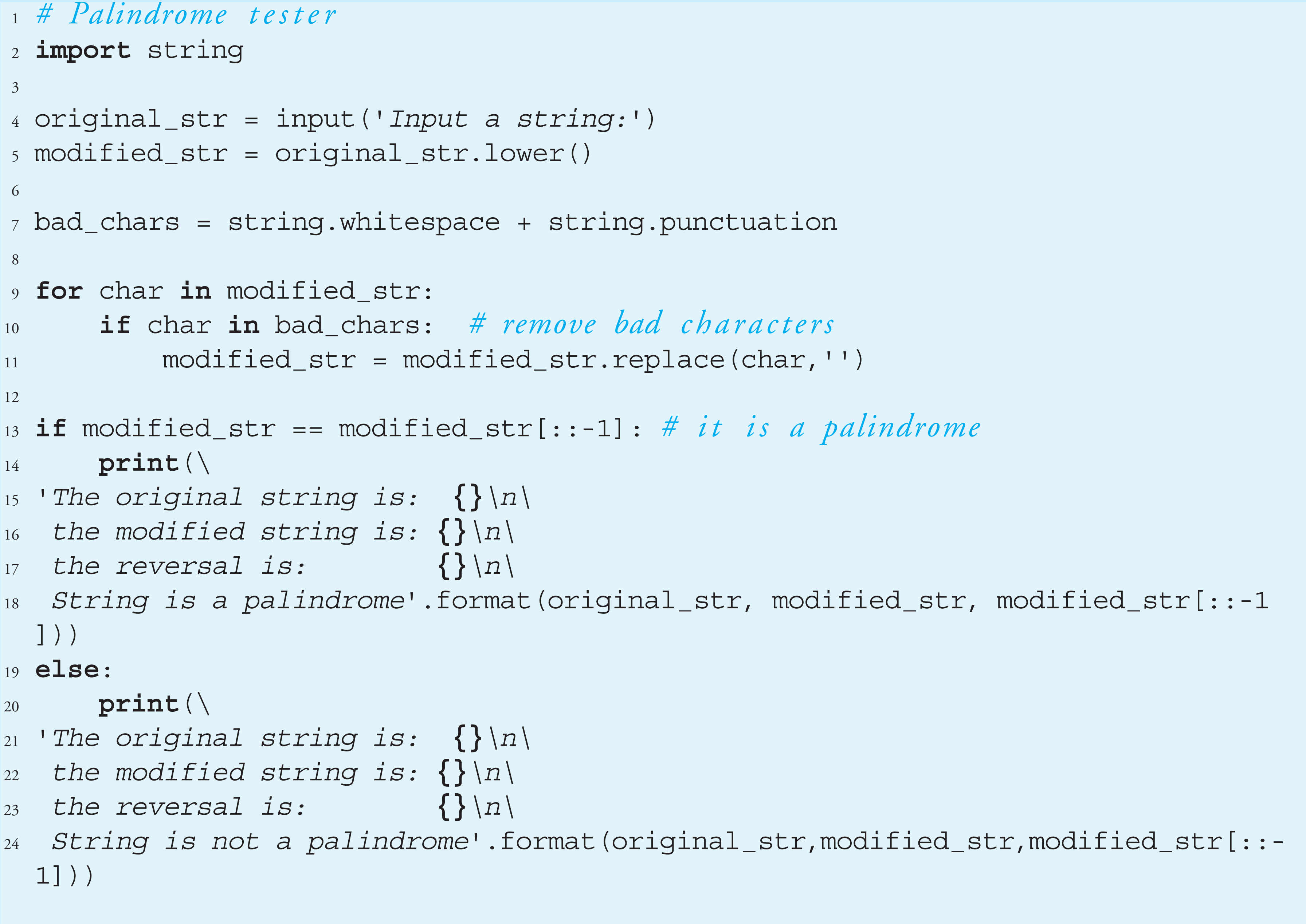
Palindromes and the rules

- A palindrome is a string that prints the same forward and backwards
- same implies that:
  - case does not matter
  - punctuation is ignored
- "Madam I'm Adam" is thus a palindrome

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
Palindromes



```python
# 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]))
```

More String Formatting

We said a format string was of the following form:

```
{:align width .precision descriptor}
```

Well, it can be more complicated than that

```
{arg : fill align sign # 0 width
, .precision descriptor}
```

That's a lot, so let's look at the details

arg

To over-ride the {}-to-argument matching we have seen, you can indicate the argument you want in the bracket

- if other descriptor stuff is needed, it goes behind the arg, separated by a :

```
>>> print('{0} is {2} and {0} is also {1}'.format('Bill',25,'tall'))
Bill is tall and Bill is also 25
```

fill, =

Besides alignment, you can fill empty spaces with a fill character:

- 0=   fill with 0's
- +=   fill with +

sign

- + means a sign for both positive and negative numbers
- - means a sign for only negative numbers
- space means space for positive, minus for negative

example

args are before the :, format after

```
>>> print('{0:.>12s} | {1:0=+10d} | {2:->5d}'.format('abc',35,22))
.........abc | +000000035 | ---22
```

for example {1:0=10d} means:

- 1→ second (count from 0) arg of format, 35
- :→ separator
- 0= →fill with 0's
- +→ plus or minus sign
- 10d→ occupy 10 spaces (left justify) decimal

# , and 0

- # is complicated, but the simple version is that it forces a decimal point 0 forces fill of zero's (equivalent to 0=)
- , put commas every three digits

```
>>> print('{:#6.0f}'.format(3)) # decimal point forced
    3.

>>> print('{:04d}'.format(4))   # zero preceeds width
0004
>>> print('{:,d}'.format(1234567890))
1,234,567,890
```

nice for tables

```
>>> for n in range(3,11):
    print('{:4}-sides:{:6}{:10.2f}{:10.2f}'.format(n,180*(n-2),180*(n-2)/n,360/n)
```

```
   3-sides:   180     60.00    120.00
   4-sides:   360     90.00     90.00
   5-sides:   540    108.00     72.00
   6-sides:   720    120.00     60.00
   7-sides:   900    128.57     51.43
   8-sides:  1080    135.00     45.00
   9-sides:  1260    140.00     40.00
  10-sides:  1440    144.00     36.00
```

Reminder, rules so far

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 imporve your programming and problem solving skills is to practice!
4. A foolish consistency is the hobgoblin of little minds
5. Test your code, often and thoroughly
6. If it was hard to write, it is probably hard to read. Add a comment.