

CS I

Introduction to Computer Programming

Lecture 2: September 30, 2015

String Theory



Caltech CS 1: Fall 2015



Last time

- Overview of course
- Introduction to Python
 - running programs
 - types, expressions, operators
 - variables and assignment
 - functions
 - using and defining with **def**



Today

- Strings
 - a fundamental data type
 - an example of an "object"
 - very widely used in practice



Terminology

- In programming, many familiar words are used with unfamiliar meanings
- Example: **string**
- Doesn't mean this:



Terminology

- In programming, many familiar words are used with unfamiliar meanings
- Example: **string**
- Instead, it means a *sequence of characters*
- Examples:
'this is a string'
"so is this"



Applications

- Strings are one of the most-used data types
- Examples:
 - DNA sequences: "**ACCTGGAACT**"
 - Web pages
 - Documents in word processors
 - Computer source code!
 - User interaction (string input/output)
 - etc.
- *Lots* of useful functions predefined for strings



Sequence...

- In Python, strings are just one example of *sequences*
- Other kinds of sequences exist
 - e.g. **lists, tuples**
- Significance:
 - the *same* functions and operators can usually be used with different sequences, and they mean the *same* kind of thing
- In other words:
 - "learn one sequence, learn them all" (almost)



... of characters

- In Python, there is no special data type for characters (letters, digits, etc.)
 - unlike many other computer languages
- A character can only be represented as a string of length 1:
 - `'a'` # the character a (letter)
 - `'1'` # the character 1 (digit)
 - `'_'` # the underscore character
 - `'?'` # question mark (symbol)



Quotation marks

- Python allows you to use either single or double quotes to represent a string:

`'I am a string'`

`"So am I"`

- You have to be consistent!

`"This is no good'`

`'This is also no good'"`

- If you start the string with a single quote, you must end it with a single quote
 - and similarly for double quotes



Quotation marks

- If you leave out the quotation marks, it isn't a string:

```
>>> 'foobar'
```

```
'foobar'
```

```
>>> foobar
```

NameError: name 'foobar' is not defined

- Python interprets **foobar** as a variable name
 - which you haven't defined, so → error



Quotation marks

- One kind of quote can be inside another:

"I can have 'single quotes' inside"

'I can have "double quotes" inside'



Quotation marks

- However, you can't put the same kind of quote inside itself:

```
s = 'this isn't going to work'
```

- Why not?

```
s = 'this isn't going to work'
```

- Type this into Python and you get:

```
s = 'this isn't going to work'
```

^

Syntax error: invalid syntax



Syntax errors

- A "syntax error" means that you broke the rules of how the language is written
- In this case:
`s = 'this isn't going to work'`
- is interpreted as:
`s = 'this isn' t going to work'`
- The `t going to work`' part doesn't make sense to Python
 - so it aborts with an error



Escape sequences

- There is a way to put the same quotation mark inside itself:

```
s = 'this won't be a problem'
```

```
s2 = "this won\"t be a problem"
```

- Character sequences beginning with the backslash character (\) are called *escape sequences*
- They change the meaning of the next character in the string



Escape sequences

- Common escape sequences:
 - \' → literal single quote character
 - (even inside a single-quoted string)
 - \\" → literal double quote character
 - (even inside a double-quoted string)
 - \\" → literal backslash character
 - \t → tab character
 - \n → newline character



Strings are immutable

- A string is a fixed, or *immutable* object
- Once you create a string, you can't change any of the letters inside the string
 - instead, you would have to create a new string

```
here = "Caltexh" # oops!
```

```
here = "Caltech" # fixed!
```



Concatenating strings

- Two strings can be combined (concatenated) by using the `+` operator:

```
>>> 'foo' + 'bar'
```

```
'foobar'
```

```
>>> s1 = 'foo'
```

```
>>> s2 = 'bar'
```

```
>>> s1 + s2
```

```
'foobar'
```



Operator overloading

- Usually, `+` used with numbers as operands
- Python sometimes uses the same operator to mean different things depending on the type of the operands:
 - `+` with numbers as operands → addition
 - `+` with strings as operands → string concatenation
- This is known as "**operator overloading**"



Empty string

- The empty string is a valid Python object
- Written as `""` or as `' '`

```
>>> 'foo' + ''
```

```
'foo'
```

```
>>> '' + 'foo'
```

```
'foo'
```



String + number?

- Can't add strings and numbers:

```
>>> 'foo' + 10
```

*TypeError: cannot concatenate
'str' and 'int' objects*

```
>>> 10 + 'foo'
```

*TypeError: unsupported operand
type(s) for +: 'int' and 'str'*

- Don't assume that Python "knows" how to do something "obvious"



String + number?

- You probably wanted

```
>>> 'foo' + 10
```

- to give

```
'foo10'
```

- There is a way to do this:

```
>>> 'foo' + str(10)
```

```
'foo10'
```

- **str** is a built-in function that converts any value into its string representation



String + number?

- How about this?

```
>>> '32' + 10
```

- Python will not convert **10** to '**10**'
 - or '**32**' to **32**
- Python does not "know" that '**32**' represents an integer
 - so doesn't output '**42**'
 - result → error as before

```
>>> '32' + '10'
```

- Result: '**3210**', not '**42**'



String + number?

- Rule of thumb:
 - Nothing gets converted to a string unless you ask it to be converted with `str`
 - However, it is OK to add different kinds of numbers

```
>>> 10 + 4.23
```

```
14.23
```

```
>>> 3.1415926 + 3
```

```
6.1415926
```

- Python converts `int` → `float` as needed



String "multiplication"

- In contrast to addition, can "multiply" strings by integers

```
>>> 'foo' * 3
```

```
'foofoofoo'
```

```
>>> 4 * 'foo'
```

```
'foofoofoofoo'
```

```
>>> 'foo' * 0
```

```
''
```

- Operator ***** is overloaded to work on strings



String "multiplication"

- Can even try weird things:

```
>>> 'foo' * (-3)
```

```
' '
```

- Were you expecting

'oofoofoof'

- ?
- Rule: When you're not sure, try it in the interpreter!



String "multiplication"

- Don't try multiplying strings by floats!

```
>>> 'foo' * 3.5
```

*TypeError: can't multiply sequence by
non-int of type 'float'*

- Half-characters not yet supported in Python ☺
- Since this doesn't make sense, Python doesn't allow it
- Similarly, multiplying strings by other strings is an error



Interlude

Caltech CS 1: Fall 2015



Admin notes

- The computer lab is in Annenberg 104
- To get access after hours, need your ID card
- If you're registered, you should have card access
- Questions? Email Jeri Chittum (jeri@cms.caltech.edu), the CMS departmental secretary and Annenberg building manager



Admin notes

- Official lab sections start *next week*
- You can still check out the lab now
- If you know some Linux, can log in to the computers, play around with them
- If not...



Friday lecture

- "Introduction to Linux"
- Friday 2-3 PM, this room
- CMS cluster lab computers use Linux, so this will be useful
- Topics: basic terminal use, managing files and directories, starting programs, etc. etc.



CS 1 Moodle web site

- Lectures are posted on the CS 1 Moodle web site
 - usually before class starts
- Please enroll in the site if you haven't already!
 - <http://moodle.caltech.edu> and follow links to "CS 001"
 - Enrollment key: **nohtyp**



CS 1 Moodle web site

- Please fill in the "Lab section signup sheet" on the course website
- Also fill in the CS 1 ombudsperson signup sheet if you want to be an ombudsperson
 - (when you know what house you're in!)
- Read the other pages on the site too!
- Especially:
 - Collaboration policies
 - CMS cluster do's and don'ts
 - Course policies/FAQ



End of Interlude

Caltech CS 1: Fall 2015



Multi-line strings

- Normally, strings in Python can only span one line
- Trying to write a string of more than one line gives an error:

```
>>> 'this is a string
```

*SyntaxError: EOL while scanning
single-quoted string*

- **EOL** means "End Of Line"
- Even if you intended to end the string on the next line, this won't work



Multi-line strings

- Multi-line strings are written using three quotes (single or double) at either end:

```
''''this is a multi-line  
string''' # 3 ' characters  
"""this, also, is a multi-  
line string"""\n# 3 " characters
```

- Other than the quoting, exactly the same as a regular string



Multi-line strings

- Multi-line strings often called "triple-quoted strings"
- Single-line strings called "single-quoted strings"
- *NOTE:* "single-quoted" doesn't mean the kind of quoted character used (' or ")
 - it means that there is only one such character to start/end the string



Multi-line strings

- Inside a triple-quoted string, can embed:
 - single or double quote characters
 - in any combination

```
>>> '''This  
... is a 'multi-line string"  
... with "" embedded '' quotes'''  
'This\nis a \'multi-line string\"\nwith  
"" embedded \"\\' quotes'
```

- When Python prints the string, it uses `\n` instead of newlines and escapes the internal `'` characters



Typical application

- A web page can be written as a single big multiline string:

```
'''<html>  
<head>  
    <title>My home page</title>  
</head>  
<body>  
    <p>Welcome to my home page!</p>  
</body>  
</html>'''
```



Typical application

- Compare to single-line string version:

```
'<html>\n' + \
'<head>\n' + \
'  <title>My home page</title>\n' + \
'</head>\n' + \
'<body>\n' + \
'  <p>Welcome to my home page!</p>\n' + \
'</body>\n' + \
'</html>'
```



Typical application

- Single-line version is much more annoying to write!
- Note: If any expression will not fit on one line, can use `\<return>` to continue it on next line (backslash (\) followed immediately by carriage return)
- Useful for big expressions, e.g.

1 + 2 + 3 + \

4 + 5 + 6 + 7 + 8

- In previous example, triple quoted string is preferable to using lots of backslash-returns



print statement

- To print a string as output from a Python program, use the **print** statement:

```
>>> print 'hello!'  
hello!
```

- Note: not the same as just entering the string '**hello!**' :

```
>>> 'hello!'  
'hello!'
```

- (This prints the quotes too)



print statement

- The important difference:
- In a file of Python code:

```
print 'hello!'
```

- will cause `hello!` to be printed, but
`'hello!'`
by itself will do nothing
- (Running code in the interactive interpreter
is not *exactly* the same as running code in a
file.)



print statement

- Escape sequences are not shown when using `print`:

```
>>> print 'hello!\ngoodbye!'  
hello!  
goodbye!
```

- Without `print`:

```
>>> 'hello!\ngoodbye!'  
'hello!\ngoodbye!'
```

- Newline character still there, but printed differently



print statement

- **print** automatically adds a newline (`\n`) character after printing:

```
>>> print 'hello'  
hello
```

- To suppress the newline character, add a comma after the statement:

```
>>> print 'hello'  
hello
```

- This changes the newline to a space character



print statement

- Example use of `,` with print in a file of Python code:

```
print 'hello'
```

```
print 'goodbye'
```

- Gives:

hello

goodbye



print statement

- Example use of `,` with print in a file of Python code:

```
print 'hello',  
print 'goodbye'
```

- Gives:

`hello goodbye`

- `,` is rarely needed in practice
- There is also a way to not even print a space after `print` (will see later in course)



print statement

- Can print multiple items:

```
>>> i = 10
```

```
>>> x = 42.3
```

```
>>> print 'i =', i, 'x =', x
```

```
i = 10 x = 42.3
```

- Adjacent items separated by a single space
- Mnemonic: comma in a **print** statement causes a space to be printed



String formatting and %

- Using comma-separated values with `print` is tedious with a lot of values
 - also inflexible → how to specify formatting?
- Better approach: use the `%` formatting operator
- `%` is another overloaded operator
 - With numeric operands, means "remainder after division"
 - With string as left-hand operand, uses string as a *format string* to create a new string



String formatting and %

- Examples of % operator in action:

```
>>> 10 % 3           ← remainder  
1  
>>> 10.4 % 3         ←  
1.4000000000000004  
>>> '2 + 2 = %d' % 4 ← formatting  
'2 + 2 = 4'
```



String formatting and %

- % operator used as a formatting operator takes
 - a format string as left operand
 - one or more variables/values as right operand
- Inside the format string:
 - special format specifiers indicate what kind of value should be substituted into string at that point
 - returns a *new* string with the substitution
 - format string is not altered!



String formatting and %

- Format specifiers:
 - `%d` means "put an int here"
 - `%f` or `%g` means "put a float here"
 - `%s` means "put a string here"



String formatting and %

- Examples:

```
>>> '2 + 2 = %d' % 4
```

```
'2 + 2 = 4'
```

```
>>> '4.3 + 5.4 = %f' % 9.7
```

```
'4.3 + 5.4 = 9.700000'
```

```
>>> '4.3 + 5.4 = %g' % 9.7
```

```
'4.3 + 5.4 = 9.7'
```

```
>>> 'hello, %s!' % 'Bob'
```

```
'hello, Bob!'
```



String formatting and %

- Note: `%s` will always work in any case!
- It will convert the right-hand argument to a string and use that for the substitution

```
>>> '2 + 2 = %s' % 4
```

```
'2 + 2 = 4'
```

```
>>> '4.3 + 5.4 = %s' % 9.7
```

```
'4.3 + 5.4 = 9.7'
```

- But: sometimes want more control over resulting string



String formatting and %

- Format modifiers:
 - a **number preceding the format specifier** gives the length of the number or string (padded with spaces to fit)
 - called the "**field width**"
 - a number preceding the format specifier and **following a decimal point** gives the number of decimal points to print (floats only)
 - called the "**precision specifier**"
 - Can have both of these



String formatting and %

- Examples:

```
>>> '1.234 + 2.345 = %.5f' % 3.579
```

3.57900 ← 5 decimal places

```
>>> '1.234 + 2.345 = %.5g' % 3.579
```

3.579 ← same number, no trailing 0s

```
>>> '1.234 + 2.345 = %.2f' % 3.579
```

3.58 ← round to 2 decimal places

```
>>> '1.234 + 2.345 = %.2g' % 3.579
```

3.6 ← round to 2 significant figures



String formatting and %

- Examples:

```
>>> 'My name is %s' % 'Mike'
```

```
'My name is Mike'
```

```
>>> 'My name is %10s' % 'Mike'
```

```
'My name is Mike' 10 characters
```

```
>>> '1.234 + 2.345 = %10.4f' % 3.579
```

```
'1.234 + 2.345 = 3.5790' 10 characters
```

```
>>> '1.234 + 2.345 = %10.4g' % 3.579
```

```
'1.234 + 2.345 = 3.579' 10 characters
```



String formatting

- Can use variables to provide values:

```
>>> i = 10
```

```
>>> 'i = %d' % i
```

```
'i = 10'
```

```
>>> x = 23.4
```

```
>>> 'x = %g' % x
```

```
'x = 23.4'
```

```
>>> s = 'I am a string'
```

```
>>> 's = %s' % s
```

```
's = I am a string'
```



String formatting

- Can use expressions to provide values:

```
>>> i = 10
```

```
>>> 'i = %d' % (i * 20)
```

```
'i = 200'
```

```
>>> x = 23.4
```

```
>>> 'x = %g' % (x / 2)
```

```
'x = 11.7'
```

```
>>> s = 'I am a string'
```

```
>>> 's = %s' % (s + " - NOT!")
```

```
's = I am a string - NOT!'
```



String formatting

- Can also format multiple items:

```
>>> i = 10
```

```
>>> x = 23.4
```

```
>>> s = 'I am a string'
```

```
>>> 'i = %d, x = %g, s = %s' % (i, x, s)
```

```
'i = 10, x = 23.4, s = I am a string'
```

- The `(i, x, s)` is not special syntax!
- It's an example of a *tuple* (collection object)
 - We'll learn more about tuples in later lectures
 - `%` used as a formatting operator knows how to use a tuple to fill in multiple values into one format string



String formatting

- Other format specifiers:
 - `%%` means a literal `%` symbol:

```
>>> 'You need to give %d%% tonight!' % 110
'You need to give 110% tonight!'
```
 - `%i` is another way to say `%d`
 - `i` means "int", `d` means "decimal"
 - Many other uncommon `%` specifiers exist



String formatting

- Most often, use `%` along with `print` to print data in a particular format:

```
>>> print 'My name is: %s.' % 'Mike'
```

```
My name is Mike.
```

```
>>> print 'pi = %10.6f' % 3.141592653589
```

```
pi = 3.141593
```

```
>>> print '%d + %d = %d' % (2, 2, 4)
```

```
2 + 2 = 4
```



Typical application (again)

- Let's use string formatting to parameterize our web page:

```
print '''\n<html>\n<head>\n    <title>%s's home page</title>\n</head>\n<body>\n    <p>Welcome to %s's home page!</p>\n</body>\n</html>''' % ('Mike', 'Mike')
```



Typical application (again)

- Results:

```
<html>
<head>
    <title>Mike's Home Page</title>
</head>
<body>
    <p>Welcome to Mike's home page!</p>
</body>
</html>
```

- Much more sophisticated web templating systems exist, but it's basically this idea



Next time

- Handling user input
- *Objects and methods*
 - First look at object-oriented programming
- How to organize our code into **modules** (AKA "libraries")
- How to document our code with **docstrings** (documentation strings)

