# Strings in Python

Pai H. Chou
National Tsing Hua University

# Outline

- Strings with Special Characters
- Raw Strings and Multi-line Strings
- Formatting strings
    - % formatting  (Python 2)
    - .format() method (Python 3)
    - f-strings (Python 3.6)
- Parsing strings by .split()

# Special characters: backslash escape

- Why? because some characters are
  - part of the language: ', "
  - can't be displayed or typed (e.g., null=ASCII 0)

| string form | ASCII | description |
|---|---|---|
| `'\0'` | 0 | null character |
| `'\t'` | 9 | tab (horizontal) |
| `'\n'` | 10 | newline |
| `'\r'` | 13 | carriage return |
| `'\\'` | 92 | backslash itself |
| `'\''` | 39 | single quote |

# Example of special characters in strings

```
>>> s = '\tEnglish\tSpanish\n\tone\tuno\n\ttwo\tdos'
>>> print(s)
        English Spanish
        one     uno
        two     dos
>>> s    # show the string literal
'\tEnglish\tSpanish\n\tone\tuno\n\ttwo\tdos'
```

- Difference between printing and having python render a *string literal*:
  - python interactive mode shows the *string literal* in valid python syntax (so it can be copy-pasted and used in another python statement)
  - when printing, the `\t`, `\n` etc get expanded

# char literal by backslash-escaped character code

- Two ways
  - octal: \ followed by 3 octal digits for ASCII code
  - hex: \x followed by 2 hex digits for ASCII code

```
>>> '%c' % 65      # ASCII 65 is 'A'
'A'
>>> oct(65)
'0o101'            # 65 decimal is 101 octal
>>> '\101'         # char for ASCII code 101 octal = 65 dec = 'A'
'A'
>>> hex(65)        # 65 decimal is 65 hex
'0x41'
>>> '\x41'         # char for ASCII code 41 hex = 65 dec = 'A'
'A'
>>>
```

# ' vs. "

- both are used for quoting strings or characters
  - be consistent: use the same for open and close
  - why two types of quotes? because you may want to quote the other without having to do escapes!
- Examples
  - `"hello, I'm John"`. Equivalent to `'hello, I\'m John.'`
  - `'she says, "This is great!" and left.'`
    Equivalent to `"she says, \"This is great!\" and left."`

# Raw strings

- syntax: `r"`*string content*`"` or
  `r'`*string content*`'`
  - Purpose: don't interpret backslash escape
    - e.g., `\n` in a raw string means `\n`, not newline

```
>>> print(r'\n means newline') # easier to type w/out escape
\n means newline
>>> print('\\n means newline') # need to protect backslash
\n means newline
>>> print(r"print('\\n means newline')") # no need to escape
print('\\n means newline')
>>> print("print('\\\\n means newline')") # each \ needs escape
print('\\n means newline')
```

# Long strings that span multiple lines

- \ at end-of-line to continue long string
  - newline is NOT part of the string literal's value
- Equivalent to multiple str literals (joined)

```
>>> s = 'hello\
... world'           # \ is a continuation, not a newline
>>> s
'helloworld'
>>> t = 'hello' 'world'  # multiple string literals are joined!
>>> t
'helloworld'
>>> u = 'hello' \
'world'
>>> u
'helloworld'
```

# Triple quotes: allow newlines

- Useful for newlines
- Plain quotes: \n and \ continuation

```
sourceCode = '''<html>
<head><title>Hi!</title></head>
<body>
<h1>Welcome to
my homepage</h1>
Good day
</body>
</html>'''
print(sourceCode)
```

```
sourceCode = '<html>\n'\
'<head><title>Hi!</title></head>\n'\
'<body>\n'\
'<h1>Welcome to\n'\
'my homepage</h1>\n'\
'Good day\n'\
'</body>\n'\
'</html>\n'
print(sourceCode)
```

# 1. Formatting Strings

- Traditional % formatting operator
- str.format() method  (Python 3)
  - by position, by index, by keyword
  - type conversion and format specifier
- f-strings  (Python 3.6)
  - embedded expressions

# string formatting with % operator

- syntax: *formatString* `%` args
  - formatString can contain `%c`, `%d`, `%s`, `%f`, etc
  - result: new string with format codes replaced

```
>>> month = 'Oct.'
>>> day = 14
>>> year = 2019
>>> 'Due %s %d, %d' % (month, day, year)
'Due Oct. 14, 2019'
>>> n = 12345678
>>> '%d is %x hex, %o octal' % (n, n, n)
'12345678 is bc614e hex, 57060516 octal'
```

# Formatting numbers

- `%d` decimal integer, `%o` octal, `%x` hex
- `%f` floating point
- `%5d` and `%05d` make it at least 5 positions wide
  - `%5d` pads empty positions on the left as needed
  - `%05d` pads empty positions with 0 on the left
- `%9.2f` and `%09.2f` make it at least 9 positions for whole number, exactly 2 positions after decimal
  - `%9.2f` pads empty positions on the left with space
  - `%09.2f` pads empty positions on the left with 0

# Examples with integer formatting

```
>>> '%8x' % 0x23     # format as 8-character hex by padding space
'      23'
>>> '%08x' % 0x23    # format as 8-char hex by padding 0 on left
00000023
>>> '%#08x' % 0x23   # format as 8-character hex including 0x
'0x000023'
>>> 0x23
35
>>> '%5d' % 123456    # the number takes more than 5 digit
'123456'
```

# Examples with float formatting

```
>>> '%9.2f' % 12.3     # 9.2 means 9 total (4 spaces, 12, ., 30)
'    12.30'
>>> '%09.2f' % 12.3
'000012.30'
>>> '%+9.2f' % 12.3    # + sign in format forces + if nonnegative
'   +12.30'
>>> '%+9.2f' % -12.3   # if number negative, still display -
'   -12.30'
>>> '%9.2f' % 12345678.3    # the .2f means .30, even though over 9
'12345678.30'
```

# Scientific notation

- floating point with 10-based exponent
- Example: `2e3`
  - means $2.0 \times 10^3 = 2000.0$
    - floating point number, even though 2 looks int
- Exponent part could be negative
  - `2e-3` = $2.0 \times 10^{-3} = 0.002$
  - `2.345e+6` = $2.345 \times 10^6 = 2345000.0$

# Formatting with scientific notation: %e

```
>>> '%e' % 2.3
'2.300000e+00'
>>> '%1.3e' % 2.3   # format as 8-char hex by padding 0 on left
'2.300e+00'
>>> '%10.3e' % 2.3
' 2.300e+00'
>>> '%+15.3e' % 245.3
'     +2.453e+02'
>>> '%+015.3e' % 245.3
'+000002.453e+02'
```

# Character formatting

- %c
  - formats an int or char (string of length 1) as a char
  - argument = the character code
- ASCII code - for American character set

| code | char |
|---|---|
| 0-31 | control characters |
| 32-47 | !"#$%&'()*+,-./ |
| 48-64 | '0'..'9' :;<=>?@ |
| 65-90 | 'A'..'Z' |

| code | char |
|---|---|
| 91-96 | [\]^_' |
| 97-122 | 'a'..'z' |
| 123-126 | {\|} |
| 127 | escape |

- Unicode - for other languages

# Character formatting examples

```
>>> '%c' % 64
'@'
>>> '%c' % 48      # 48 is ASCII for '0'
'0'
>>> '%3c' % 78     # 78 is ASCII for 'N'
'  N'
>>> '%c' % 'Z'
'Z'
>>> '%c' % 'ZZ'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: %c requires int or char
```

# 2. str .format() method

- Recommended for Python3
- Idea: use {} inside format string to specify items for substitution
- Can be addressed
  - by position
  - by explicit index
  - by name
- Can take formats or use default representation

# 2.1 By position

```
>>> s = 'Hello {}, your ID is {}'
>>> s.format('Harry', 12345)
'Hello Harry, your ID is 12345'
```

- s has two `{}` slots to fill
- `.format()` method fills the slots with params
  - `'Harry'` goes to the left `{}`
  - `12345` fills the right `{}`

# 2.2 By position index

```
>>> s = 'Hello {0}, your ID is {1}'
>>> s.format('Harry', 12345)
'Hello Harry, your ID is 12345'
```

- The `{}` can have the index starting from 0

- Why useful? Allows repeating substitution

```
>>> t = 'one {0}, two {0}s, three {0}s'
>>> t.format('apple')
'one apple, two apples, three apples'
```

  - in `%s` style, would have to pass `'apple'` 3 times

```
>>> 'one %s, two %ss, three %ss' % ('apple', 'apple', 'apple')
'one apple, two apples, three apples'
```

# 2.3 By keyword argument

```
>>> u = 'Hello {name}, your ID is {id}'
>>> u.format(name='Harry', id=12345)
'Hello Harry, your ID is 12345'
```

- The {} can enclose a keyword
- keyword=value parameter passing is nice
  - easier to read and understand
  - less likely to make mistakes compared to position based or index based arguments

# str.format() conversion type

- `{:s}` str, `{:d}` dec, `{:x}` hex, `{:f}` float
  - similar to `%s`, `%d`, `%x`, `%f`, etc.
  - space padding and 0 padding also work

```
>>> v = 'Hello {name:s}, your ID is {id:08d}'
>>> v.format(name='Harry', id=12345)
'Hello Harry, your ID is 00012345'
```

  - `{id:08d}` means pad 0s on left to make 8 digits
- works with positional or index arguments!

```
>>> w = 'Hello {:s}, your ID is {:08d}'
>>> w.format('Harry', 12345)
'Hello Harry, your ID is 00012345'
```

# str.format() conversion type

- works with positional arguments

```
>>> w = 'Hello {:s}, your ID is {:08d}'
>>> w.format('Harry', 12345)
'Hello Harry, your ID is 00012345'
```

- works with index arguments

```
>>> v = 'Hello {0:s}, your ID is {1:08d}'
>>> v.format('Harry', 12345)
'Hello Harry, your ID is 00012345'
```

# Padding with Alignment

- :< left-align, :> right-align, :^ centered

```
>>> x = 'Hello {name:<20s}, your ID is {id:08d}'
>>> x.format(name='Harry', id=12345)
'Hello Harry                , your ID is 00012345'
>>> y = 'Hello {name:>20s}, your ID is {id:08d}'
>>> y.format(name='Harry', id=12345)
'Hello                Harry, your ID is 00012345'
```

- Padding with non-space characters
  - put the padding char before alignment char
  - e.g., :_^20s center-aligns the string and pads _

```
>>> z = 'Hello {name:_^20s}, your ID is {id:08d}'
>>> z.format(name='Harry', id=12345)
'Hello _______Harry________, your ID is 00012345'
```

# Example: multiplication table first attempt

- first attempt as a function

```
def MultTable(L, R):
    '''L and R are ranges'''
    for left in L:
        for right in R:
            print('{} x {} = {}'.format(left, right, left*right))
if __name__ == '__main__':
    lRange = range(9,12)
    rRange = range(8,11)
    MultTable(lRange, rRange)
```

```
$ python3 mult.py
9 x 8 = 72
9 x 9 = 81
9 x 10 = 90
10 x 8 = 80
10 x 9 = 90
10 x 10 = 100
11 x 8 = 88
11 x 9 = 99
11 x 10 = 110
```

- output: not aligned!
  - want int to be right-aligned
  - how? specify #positions like {:3d}

# Example: multiplication table second attempt

- specify #positions like {:3d}

```
def MultTable(L, R):
    '''L and R are ranges'''
    for a in L:
        for b in R:
            print('{:3d} x {:3d} = {:3d}'.format(a, b, a*b))
if __name__ == '__main__':
    lRange = range(9,12)
    rRange = range(8,11)
    MultTable(lRange, rRange)
```

```
$ python3 mult.py
  9 x   8 =  72
  9 x   9 =  81
  9 x  10 =  90
 10 x   8 =  80
 10 x   9 =  90
 10 x  10 = 100
 11 x   8 =  88
 11 x   9 =  99
 11 x  10 = 110
```

- Issue:
  - not aligned for range with more than 3 digits...
- Solution:
  - generate the format based on max # of digits

# 3. f-strings

- string literals that start with `f` or `F`
  - `f"this is an f-string"` # uses double quotes
  - `f'this is also an f-string'` # single quotes
- What does it do?
  - evaluates the content of {} as expression!
- Why?
  - can be easier to read

# f-string example

- f-string: evaluate expression in {}

```
>>> name='Harry'; id=12345
>>> f'Hello {name}, your ID is {id}'
'Hello Harry, your ID is 12345'
```

  - easier to read, expression is inlined

- compared to % and `.format()`

```
name='Harry'; id=12345
'Hello %s, your ID is %d' % (name, id)
'Hello {}, your ID is {}'.format(name, id)
'Hello {0}, your ID is {1}'.format(name, id)
'Hello {name}, your ID is {id}'.format(name=name, id=id)
```

```
f'Hello {name}, your ID is {id}'
```

  - f-string is shortest, most concise!

# f-strings can take expressions

- you can perform computation

```
>>> x = 10; y = 20
>>> f'{x} + {y} = {x+y}, {x} - {y} = {x-y}'
'10 + 20 = 30, 10 - 20 = -10'
```

- you can call function or method!

```
>>> L = ['a', 'b', 'c']
>>> f'L = {L}, len(L) = {len(L)}'
"L = ['a', 'b', 'c'], len(L) = 3"
```

# Conversion specifiers are same as .format() ones

- `{:s}` str, `{:d}` dec, `{:x}` hex, `{:f}` float
  - {:08d} to pad 0's on right, 8 positions minimum

```
>>> name='Harry'; id=12345
>>> f'Hello {name}, your ID is {id:08d}'
'Hello Harry, your ID is 00012345'
```

# Summary: string formatting

- old % formatting (all versions)
  - substitutes %d, %x, %c, %s, %f, .. in format string
  - similar to C language, compatible with Python 2
- str.format() method  (Python 3 or later)
  - substitutes {} in format string
  - by position, index, or keyword
- f-strings (Python 3.6 or later)
  - substitutes {} in format string with expressions

# String splitting on blanks

- use built-in str`.split()` method
  - by default, will split on one or more "blanks" (including tabs, spaces, newlines, etc)
  - discards empty strings

```
>>> s = "   hello\t\n my   name is  Harry   "
>>> s.split()
['hello', 'my', 'name', 'is', 'Harry']
```

# String splitting on other separators

- str.`split()` to split on other characters
- e.g., use comma as separator => CSV! (comma-separated values) can export from Excel

```
>>> s = 'field 1,field 2,,value of field 3'
>>> s.split(',')
['field 1', 'field 2', '', 'value of field 3']
```

  - issue: if your data contains comma, then need to quote it
  - => simple splitting won't work; need parsing
  - may get empty string
- tab-separated values (`'\t'`) = TSV

# application: unix utility
# wc (word count)

```
$ wc mult.py
       9      32     249 mult.py
```

- 9 lines, 32 words, 249 characters

- Implementation in Python
  - follow template for arg, open textfile, readlines()
  - lines = count number of lines
  - words = split each line by blanks and count
  - character count = #chars on all lines

# Complete source for `wc` based on template code

```
#!/usr/bin/env python3
import sys
numberOfArgs = len(sys.argv)
if numberOfArgs != 2:
    sys.stderr.write('Usage: %s inputFile\n' % sys.argv[0])
    sys.exit(1)
try:
    fh = open(sys.argv[1], 'r')
except:
    sys.stderr.write('cannot open input file %s\n' % sys.argv[1])
    sys.exit(2)
lines = words = chars = 0  # initialize counters for line, word, char
for line in fh.readlines():
    lines += 1                      # add 1 for each line
    words += len(line.split()) # count num of words separated by blanks
    chars += len(line)              # increment by number of chars in line
fh.close()
print('{:8d}{:8d}{:8d} {}'.format(lines, words, chars, sys.argv[1]))
```

# String joining

- inverse operation of *str*`.split()`
- syntax: *str*`.join(`*iterable*`)`

```
>>> '..'.join(['hello', 'world', 'goodbye'])
'hello..world..goodbye'
>>> '_'.join(['hello', 'world', 'goodbye'])
'hello_world_goodbye'
```

- how it works: a list is iterable
- think for-loop, get one item at a time

# String joining by character

- syntax: *str*`.join(`*iterable*`)`
  - a string is also iterable: one char at a time
  - str.join(string)  => joins characters

```
>>> '-'.join('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
'A-B-C-D-E-F-G-H-I-J-K-L-M-N-O-P-Q-R-S-T-U-V-W-X-Y-Z'
```

# Can combine split and join

- Suppose: want to replace blanks with _

```
>>> def MakeSnakeCase(s):
...     return '_'.join(s.split())
...
>>> MakeSnakeCase('this is a name')
'this_is_a_name'
>>>
```

- how it works:
  - s.split() evaluates to a list of strings separated by blanks
  - '_'.join(...) joins the list of strings (from s.split()) by the '_' character.

# Related module: `string`

- `import string`
- `help(string)`
  - defines constants including `ascii_letters`, `ascii_lowercase`, `ascii_uppercase`, `digits`, `hexdigits`, `octdigits`, `printable`, `punctuation`, `whitespace`

```
>>> import string
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> ',' in string.punctuation
True
```

# string module

- can help with string related features
  - function `capwords()`

```
>>> import string
>>> string.capwords('hello world')
'Hello World'
```

- example: make camel case

```
>>> def MakeCamelCase(s)
...     return ''.join(string.capwords(s).split())
...
>>> MakeCamelCase('this is a test')
'ThisIsATest'
```

  - how it works: capitalize, split, then join by ''

# Summary: strings in Python

- string literals
  - backslash escape, multiline strings, raw strings
- string formatting
  - traditional % formatting, .format() method, f-strings
- string processing
  - str.split(sep) to split on separator
  - sep.join(iterable) to join strings by sep