Strings, Exceptions, Unicode, File Processing

Christopher Barker

IRIS

October 16, 2013

Table of Contents

repr vs. str

repr() vs str()

```
In [200]: s = "a string\nwith a newline"

In [203]: print str(s)
a string
with a newline

In [204]: print repr(s)
’a string\nwith a newline’
```

repr vs. str

```
eval(repr(something)) == something
```

```
In [205]: s2 = eval(repr(s))

In [206]: s2
Out[206]: 'a string\nwith a newline'
```

Strings

A string literal creates a string type

```
"this is a string"
```

Can also use `str()`

```
In [256]: str(34)
Out[256]: ’34’
```

or ”back ticks”

```
In [258]: ‘34‘
Out[258]: ’34’
```

(demo)

The String Type

Lots of nifty methods:

```
s.lower()
s.upper()
    ...
s.capitalize()
s.swapcase()
s.title()
```

http://docs.python.org/library/stdtypes.html#index-23

The String Type

Lots of nifty methods:

```
x in s
s.startswith(x)
s.endswith(x)
...
s.index(x)
s.find(x)
s.rfind(x)
```

http://docs.python.org/library/stdtypes.html#index-23

The String Type

Lots of nifty methods:

```
s.split()
s.join(list)
...
s.splitlines()
```

`http://docs.python.org/library/stdtypes.html#index-23`

Joining Strings

The Join Method:

```
In [289]: t = ("some", "words","to","join")

In [290]: " ".join(t)
Out[290]: 'some words to join'

In [291]: ",".join(t)
Out[291]: 'some,words,to,join'

In [292]: "".join(t)
Out[292]: 'somewordstojoin'
```

(demo – join)

The string module

Lots of handy constants, etc.

```
string.ascii_letters
string.ascii_lowercase
string.ascii_uppercase
string.letters
string.hexdigits
string.whitespace
string.printable
string.digits
string.punctuation
```

(and the string methods – legacy)

http://docs.python.org/2/library/string.html#module-string

String Literals

Common Escape Sequences

`\\`	Backslash (\)
`\a`	ASCII Bell (BEL)
`\b`	ASCII Backspace (BS)
`\n`	ASCII Linefeed (LF)
`\r`	ASCII Carriage Return (CR)
`\t`	ASCII Horizontal Tab (TAB)
`\ooo`	Character with octal value ooo
`\xhh`	Character with hex value hh

(http://docs.python.org/release/2.5.2/ref/strings.html)

Raw Strings

Escape Sequences Ignored

```
In [408]: print "this\nthat"
this
that
In [409]: print r"this\nthat"
this\nthat
```

Gotcha:

```
In [415]: r"\"
SyntaxError: EOL while scanning string literal
```

(handy for regex, windows paths...)

Character Values

Characters in strings are stored as numeric values

“ASCII” values: 1-127

“ANSI” values: 1-255

To get the value:

```
In [109]: for i in ’Chris’:
   .....:     print ord(i),
67 104 114 105 115

In [110]: for i in (67,104,114,105,115):
   .....:     print chr(i),
C h r i s
```

(later: unicode!)

Building Strings

Please don't do this:

```
'Hello ' + name + '!'
```

(much)

Building Strings

Do this instead:

```
'Hello %s!' % name
```

much faster and safer:

easier to modify as code gets complicated

`http://docs.python.org/library/stdtypes.html#string-formatting-operations`

String Formatting

The string format operator: %

```
In [261]: "an integer is: %i"%34
Out[261]: 'an integer is: 34'

In [262]: "a floating point is: %f"%34.5
Out[262]: 'a floating point is: 34.500000'

In [263]: "a string is: %s"%"anything"
Out[263]: 'a string is: anything'
```

String Formatting

multiple arguments:

```
In [264]: "the number %s is %i"%('five', 5)
Out[264]: 'the number five is 5'

In [266]: "the first 3 numbers are: %i, %i, %i"%(1,2,3)
Out[266]: 'the first 3 numbers are: 1, 2, 3'
```

String formatting

Gotcha

```
In [127]: "this is a string with %i formatting item"%1
Out[127]: 'this is a string with 1 formatting item'

In [128]: "string with %i formatting %s: "%2, "items"
TypeError: not enough arguments for format string

# Done right:
In [131]: "string with %i formatting %s"%(2, "items")
Out[131]: 'string with 2 formatting items'

In [132]: "string with %i formatting item"%(1,)
Out[132]: 'string with 1 formatting item'
```

String formatting

Named arguments

```
’Hello %(name)s!’%{’name’:’Joe’}
’Hello Joe!’

’Hello %(name)s, how are you, %(name)s!’ %{’name’:’Joe’}
’Hello Joe, how are you, Joe!’
```

That last bit is a dictionary (next week)

String formatting

The format operator works with string variables, too:

```
In [45]: s = "%i / %i = %i"

In [46]: a, b = 12, 3

In [47]: s%(a, b, a/b)
Out[47]: '12 / 3 = 4'
```

So you can dynamically build a format string

Advanced Formatting

The format method

```
In [14]: ’Hello {0} {1}!’.format(’Joe’, ’Barnes’)
Out[14]: ’Hello Joe Barnes!’

In [12]: ’Hello {name}!’.format(name=’Joe’)
Out[12]: ’Hello Joe!’
```

pick one (probably regular string formatting):
– get comfy with it

LAB

Fun with strings

- Rewrite:

  ```
   the first 3 numbers are: %i, %i, %i"%(1,2,3)
  ```

  for an arbitrary number of numbers...
- write a format string that will take:

  ```
  ( 2, 123.4567, 10000)
  ```

  and produce:

  ```
  'file_002 :   123.46, 1e+04'
  ```
- Write a (really simple) mail merge program
- ROT13 – see next slide

`http://docs.python.org/library/stdtypes.html#string-formatting-operations`

LAB

ROT13 encryption

Applying ROT13 to a piece of text merely requires examining its alphabetic characters and replacing each one by the letter 13 places further along in the alphabet, wrapping back to the beginning if necessary

- Implement rot13 decoding
- decode this message:

  Zntargvp sebz bhgfvqr arne pbeare

  (from a geo-caching hint)

Follow Up

Recommended Reading:

- Think Python: Chapt. 9 – 14
- Dive Into Python: Chapt. 6
- String methods: `http://docs.python.org/library/stdtypes.html#string-methods`
- Extra: unicode: `http://www.joelonsoftware.com/articles/Unicode.html`

Do:

- Finish the LABs
- Some CodingBat exercises.
- LPTHW: for extra practice with the concepts – some of: excercises 5 – 14

Files

Text Files

```
f = open('secrets.txt')
secret_data = f.read()
f.close()
```

`secret_data` is a string

(can also use `file()` – `open()` is preferred)

Files

Binary Files

```
f = open('secrets.txt', 'rb')
secret_data = f.read()
f.close()
```

`secret_data` is still a string

(with arbitrary bytes in it)

(See the `struct` module to unpack binary data)

Files

File Opening Modes

```
f = open('secrets.txt', [mode])
```

```
'r', 'w', 'a'
'rb', 'wb', 'ab'
r+, w+, a+
r+b, w+b, a+b
U
U+
```

Gotcha – w mode always clears the file

Text File Notes

Text is default

- Newlines are translated: `\r\n -> \n`
- – reading and writing!
- Use *nux-style in your code: `\n`
- Open text files with `'U'` "Universal" flag

Gotcha:

- no difference between text and binary on *nix
    - breaks on Windows

File Reading

Reading Part of a file

```
header_size = 4096

f = open('secrets.txt')
secret_data = f.read(header_size)
f.close()
```

File Reading

Common Idioms

```
for line in open('secrets.txt'):
    print line

f = open('secrets.txt')
while True:
    line = f.readline()
    if not line:
        break
    do_something_with_line()
```

File Writing

```
outfile = open('output.txt', 'w')

for i in range(10):
    outfile.write("this is line: %i\n"%i)
```

File Methods

Commonly Used Methods

```
f.read() f.readline()  f.readlines()

f.write(str) f.writelines(seq)

f.seek(offset)   f.tell()

f.flush()

f.close()
```

File Like Objects

File-like objects

Many classes implement the file interface:

- loggers
- `sys.stdout`
- `urllib.open()`
- pipes, subprocesses
- StringIO

`http://docs.python.org/library/stdtypes.html#bltin-file-objects`

StringIO

StringIO

```
In [417]: import StringIO
In [420]: f = StringIO.StringIO()

In [421]: f.write("somestuff")

In [422]: f.seek(0)

In [423]: f.read()
Out[423]: 'somestuff'
```

handy for testing

Unicode

I hope you all read this:

The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)

`http://www.joelonsoftware.com/articles/Unicode.html`

If not – go read it!

Unicode

Everything is Bytes

If it's on disk or on a network, it's bytes

Python provides some abstractions to make it easier to deal with bytes

Unicode

Unicode is a biggie

strings vs unicode
(`str()` vs. `unicode()`)

python 2.x vs 3.x

(actually, dealing with numbers rather than bytes is big – but we take that for granted)

Unicode

Strings are sequences of bytes

Unicode strings are sequences of platonic characters

Platonic characters cannot be written to disk or network!

(ANSI – one character == one byte – so easy!)

Unicode

the `unicode` object lets you work with characters

encoding is converting from a uncode object to bytes

decoding is converting from bytes to a unicode object

Unicode

```
import codecs
ord()
chr()
unichr()
str()
unicode()
encode()
decode()
```

Unicode Literals

1) Use unicode in your source files:

```
# -*- coding: utf-8 -*-
```

2) escape the unicode characters

```
print u"The integral sign: \u222B"
print u"The integral sign: \N{integral}"
```

lots of tables of code points online:
`http://inamidst.com/stuff/unidata/`

Unicode

Use unicode objects in all your code

decode on input

encode on output

Many packages do this for you
(XML processing, databases, ...)

Gotcha:
Python has a default encoding (usually ascii)

Unicode

Python Docs Unicode HowTo:

`http://docs.python.org/howto/unicode.html`

“Reading Unicode from a file is therefore simple:”

```
import codecs
f = codecs.open('unicode.rst', encoding='utf-8')
for line in f:
    print repr(line)
```

Unicode LAB

- Find some nifty non-ascii characters you might use.
  Create a unicode object with them in two different ways.
- In the "code" dir for this week, there are two files:
  `text.utf16`
  `text.utf32`
  read the contents into unicode objects
- write some of the text from the first exercise to file.
- read that file back in.

(reference: `http://inamidst.com/stuff/unidata/`)
NOTE: if you terminal does not support unicode – you'll get an error trying to print. Try a different terminal or IDE, or google for a solution

Exceptions

Another Branching structure:

```
try:
    do_something()
    f = open('missing.txt')
    process(f)   # never called if file missing
except IOError:
    print "couldn't open missing.txt"
```

Exceptions

Never Do this:

```
try:
    do_something()
    f = open('missing.txt')
    process(f)   # never called if file missing
except:
    print "couldn't open missing.txt"
```

Exceptions

Use Exceptions, rather than your own tests
– Don't do this:

```
do_something()
if os.path.exists('missing.txt'):
    f = open('missing.txt')
    process(f)    # never called if file missing
```

It will almost always work – but the almost will drive you crazy

Exceptions

"easier to ask forgiveness than permission"

– Grace Hopper

`http://www.youtube.com/watch?v=AZDWveIdqjY`

Exceptions

For simple scripts, let exceptions happen

Only handle the exception if the code can and will do something about it

(much better debugging info when an error does occur)

Exceptions – finally

```
try:
    do_something()
    f = open('missing.txt')
    process(f)   # never called if file missing
except IOError:
    print "couldn't open missing.txt"
finally:
    do_some_clean-up
```

the `finally:` clause will always run

Exceptions – else

```
try:
    do_something()
    f = open(’missing.txt’)
except IOError:
    print "couldn’t open missing.txt"
else:
    process(f) # only called if there was no exception
```

Advantage:
you know where the Exception came from

Exceptions – using them

```
try:
    do_something()
    f = open(’missing.txt’)
except IOError as the_error:
    print the_error
    the_error.extra_info = "some more information"
    raise
```

Particularly useful if you catch more than one exception:

```
except (IOError, BufferError, OSError) as the_error:
    do_something_with (the_error)
```

Raising Exceptions

```
def divide(a,b):
    if b == 0:
        raise ZeroDivisionError("b can not be zero")
    else:
        return a / b
```

when you call it:

```
In [515]: divide (12,0)

ZeroDivisionError: b can not be zero
```

Built in Exceptions

You can create your own custom exceptions
But...

```
exp = \
 [name for name in dir(__builtin__) if "Error" in name]

len(exp)
32
```

For the most part, you can/should use a built in one

Paths

Relative paths:

```
secret.txt
./secret.txt
```

Absolute paths:

```
/home/chris/secret.txt
```

Either work with `open()`, etc.

(working directory only makes sense with command-line programs...)

os.path

```
os.getcwd() -- os.getcwdu()
chdir(path)

os.path.abspath()
os.path.relpath()
```

os.path

```
os.path.split()
os.path.splitext()
os.path.basename()
os.path.dirname()
os.path.join()
```

(all platform independent)

directories

```
os.listdir()
os.mkdir()

os.walk()
```

(higher level stuff in `shutil` module)

LAB

- write a program which prints the full path to all files in the current directory, one per line
- write a program which copies a file from a source, to a destination (without using shutil, or the OS copy command)
- update mail-merge from the previous lab to write output to individual files on disk

Follow Up

- TP: Chapters – 10, 11, 12, 13
- Finish (or re-factor) the Labs you didn't finish in class.
- CodingBat - 12 more string & list problems
- Write a script which does something useful (to you) and reads & writes files. Very, very small scope is good. something useful at work would be great.

Dictionary

Python calls it a `dict`

Other languages call it:

- dictionary
- associative array
- map
- hash table
- hash
- key-value pair

Dictionary Constructors

```
>>> {’key1’: 3, ’key2’: 5}
{’key1’: 3, ’key2’: 5}

>>> dict([(’key1’, 3),(’key2’, 5)])
{’key1’: 3, ’key2’: 5}

>>> dict(key1=3, key2= 5)
{’key1’: 3, ’key2’: 5}

>>> d = {}
>>> d[’key1’] = 3
>>> d[’key2’] = 5
>>> d
{’key1’: 3, ’key2’: 5}
```

Dictionary Indexing

```
>>> d = {’name’: ’Brian’, ’score’: 42}
>>> d[’score’]
42
>>> d = {1: ’one’, 0: ’zero’}
>>> d[0]
’zero’
>>> d[’non-existing key’]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: ’non-existing key’
```

Dictionary Indexing

Keys can be any immutable:

- numbers
- string
- tuples

```
In [325]: d[3] = 'string'
In [326]: d[3.14] = 'pi'
In [327]: d['pi'] = 3.14
In [328]: d[ (1,2,3) ] = 'a tuple key'
In [329]: d[ [1,2,3] ] = 'a list key'
   TypeError: unhashable type: 'list'
```

Actually – any "hashable" type.

Dictionary Indexing

hash functions convert arbitrarily large data to a small proxy (usually int)

always return the same proxy for the same input

MD5, SHA, etc

Dictionary Indexing

Dictionaries hash the key to an integer proxy and use it to find the key and value

Key lookup is efficient because the hash function leads directly to a bucket with a very few keys (often just one)

Dictionary Indexing

What would happen if the proxy changed after storing a key?

Hashability requires immutability

Dictionary Indexing

Key lookup is very efficient

Same average time regardless of size

also... Python name look-ups are implemented with dict:
— its highly optimized

Dictionary Indexing

key to value
lookup is one way

value to key
requires visiting the whole dict

if you need to check dict values often, create another dict or set (up to you to keep them in sync)

Dictionary Ordering (not)

dictionaries have no defined order

```
In [352]: d = {’one’:1, ’two’:2, ’three’:3}

In [353]: d
Out[353]: {’one’: 1, ’three’: 3, ’two’: 2}

In [354]: d.keys()
Out[354]: [’three’, ’two’, ’one’]
```

Dictionary Iterating

for iterates the keys

```
>>> d = {'name': 'Brian', 'score': 42}
>>> for x in d:
...   print x
...
score name
```

note the different order...

dict keys and values

```
>>> d.keys()
['score', 'name']

>>> d.values()
[42, 'Brian']

>>> d.items()
[('score', 42), ('name', 'Brian')]
```

dict keys and values

iterating on everything

```
>>> d = {'name': 'Brian', 'score': 42}
>>> for k, v in d.items():
...     print "%s: %s" % (k, v)
...
score: 42
name: Brian
```

Dictionary Performance

- indexing is fast and constant time: O(1)
- x in s cpnstant time: O(1)
- visiting all is proportional to n: O(n)
- inserting is constant time: O(1)
- deleting is constant time: O(1)

http://wiki.python.org/moin/TimeComplexity

Dict Comprehensions

You can do it with dicts, too:

```
new_dict = { key:value for variable in a_sequence}
```

same as for loop:

```
new_dict = {}
for key in a_list:
    new_dict[key] = value
```

Dict Comprehensions

Example

```
In [340]: { i: "this_%i"%i for i in range(5) }
Out[340]: {0: 'this_0', 1: 'this_1', 2: 'this_2',
           3: 'this_3', 4: 'this_4'}
```

(not as useful with the dict() constructor...)

Switch ?

How do you spell switch/case in Python?

Put the values to switch on in the keys:

Functions to call in values:

demo: sample code (`switch_case.py`)

Sets

`set` is an unordered collection of distinct values

Essentially a dict with only keys

Set Constructors

```
>>> set()
set([])
>>> set([1, 2, 3])
set([1, 2, 3])
# as of 2.7
>>> {1, 2, 3}
set([1, 2, 3])
>>> s = set()
>>> s.update([1, 2, 3])
>>> s
set([1, 2, 3])
```

Set Properties

Set members must be hashable

Like dictionary keys – and for same reason (efficient lookup)

No indexing (unordered)

```
>>> s[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
```

Set Methods

```
>> s = set([1])
>>> s.pop() # an arbitrary member
1
>>> s.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: ’pop from an empty set’

>>> s = set([1, 2, 3])
>>> s.remove(2)
>>> s.remove(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 2
```

Set Methods

```
s.isdisjoint(other)

s.issubset(other)

s.union(other, ...)

s.intersection(other, ...)

s.difference(other, ...)

s.symmetric_difference( other, ...)
```

Frozen Set

Also `frozenset`

immutable – for use as a key in a dict
(or another set...)

```
>>> fs = frozenset((3,8,5))
>>> fs.add(9)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

Function arguments in variables

function arguments are really just
– a tuple (positional arguments)
– a dict (keyword arguments)

```
def f(x, y, w=0, h=0):
    print "position: %s, %s -- shape: %s, %s"%(x, y, w, h)

position = (3,4)
size = {’h’: 10, ’w’: 20}

>>> f( *position, **size)
position: 3, 4 -- shape: 20, 10
```

Function parameters in variables

You can also pull in the parameters out in the function as a tuple and a dict

```
def f(*args, **kwargs):
    print "the positional arguments are:", args
    print "the keyword arguments are:", kwargs

In [389]: f(2, 3, this=5, that=7)
the positional arguments are: (2, 3)
the keyword arguments are: {’this’: 5, ’that’: 7}
```

LAB

dict LAB:

`code/dict_set_excercises.rst`

or

Optional LAB

- Coding Kata 14 - Dave Thomas `http://codekata.pragprog.com/2007/01/kata_fourteen_t.html`
- See how far you can get on this task using The Adventures of Sherlock Holmes as input: sherlock.txt in the week04 directory (ascii)
- This is intentionally open-ended and underspecified. There are many interesting decisions to make.

Follow Up

- Spend more time (or some time) with the Coding Kata from lab. Get it basically working.
- Experiment with different lengths for the lookup key. (3 words, 4 words, 3 letters, etc)
- This assignment is about playing around with the algorithm and data.