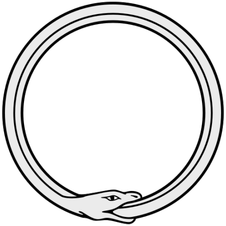
CS 3101-3 - Programming Languages: Python

Lecture 2: Strings/IO/Functions

Daniel Bauer (bauer@cs.columbia.edu)

September 17 2014



Contents

else in loops

- while and for loops, that contain a break statement can contain an optional else clause.
- The else block is executed after the loop finishes normally, but NOT when it is terminated by break.

```
for n in range(2, 10):
    for x in range(2, n):
        # check if x is a factor of
            n
        if n % x == 0:
            print(n, '=', x, '*', n
                /x)
            break
        else:
          # didn't find a factor
              for n
          print(n, 'is prime')
```

```
$python prime.py
2 is prime
3 is prime
4 = 2 * 2
5 is prime number
6 = 2 * 3
7 is prime
8 = 2 * 4
9 = 3 * 3
```

Strings

Files and IO

Functions

String literals (1)

- String literals can be defined with single quotes or double quotes.
- Can use other type of quotes inside the string.

```
>>> str = 'Hello "World"'
```

- Can use ''' or """ to delimit multi-line strings.

```
>>> s = """Hello
    "World"
"""
>>> print(s)
Hello
"World"
```

String literals (2)

- Some characters need to be 'escaped'.

```
>>> print('Hello \'world\"')
Hello 'world"
>>> print('Hello \\ World') # Backslash
Hello \ World
>>> print('Hello \n World') # Line feed
Hello
 World
>>> print('Hello\t World') # Tab
Hello	 World
```

String Operations - Review

- Strings support all sequence operations.

```
>>> len('foo')  # Get length
3
>>> 'a' * 10 + 'rgh' # Concatenation and repition
'aaaaaaaaaargh'
>>> 'tuna' in 'fortunate' # Substring
True
>>> 'banana'.count('an') # Count substrings
2
>>> 'banana'.index('na') # Find index
2
>>> 'banana'[2:-1]  # slicing
'nan'
```

- Also iteration and list comprehension.

Additional String Operations (1)

- Capitalize first letter, convert to upper/lower case or Title Case.

```
>>> 'grail'.capitalize()
'Grail'
>>> 'grail'.upper()
'GRAIL'
>>> 'GRAIL'.lower()
'grail'
>>> 'the holy grail'.title()
'The Holy Grail'
```

- Check whether the string begins or starts with a string.

```
>>> "python".startswith("py")
True
>>> "python".endswith("ython")
True
```

Additional String Operations (2)

- Split a string into a list of its components using a separator

```
>>> "python, java, lisp, haskell".split(", ")
['python', 'java', 'lisp', 'haskell']

>>> #Default: runs of whitespaces, tabs, linefeeds
... "An African\t or European\n swallow?".split()
['An', 'African', 'or', 'European', 'swallow?']
```

- Join together a sequence of strings using a seperator string.

```
>>> # Format a list in CSV format:
>>> ', '.join(['Galahad', 'the pure', 'yellow'])
'Galahad, the pure, yellow'
```

Additional String Operations (3)

- Certain simple tests on strings:
  - contains only digits?

```
>>> '42'.isdigit()
True
```

  - contains only upper/lowercase letters?

```
>>> 'Alpha'.isalpha()
True
```

  - contains only upper/lowercase letters?

```
>>> '535mudd'.isalnum()
True
```

Regular expressions provide more advanced testing.

String Formatting (1)

- Often used to pretty-print data or to write it to a file.
- `formatstr.format(argument_0, argument_1 ...)` replaces placeholders in formatstr with arguments.
- Placeholder {i} is replaced with the argument with index *i*.

```
>>> "{0}, {1}C, Humidity: {2}%".format('New York',
     10.0, 48)
'New York, 10.0C, Humidity: 48%'
>>> # Can assign names to format fields
... "{temp}C".format('New York', 48, temp=10.0)
'10.0C'
>>> #Literal { need to be escaped by duplication.
... "{{ {temp}C }}".format(temp=10.0)
'{ 10.0C }'
```

- Arguments are implicitly converted to `str`.

String Formatting (2)

- If an argument is a sequence, can use indexing in format string.

```
"{0[0]}, {0[1]}, and {0[2]}".format(('a','b','c'))
```

- Placeholders can contain format specifiers (after a :).
  - e.g specify minimum field with and set alignment

```
>>> "|{0:^5}|{0:<5}|{0:>5}|".format("x", "y", "z")
'|  x  |x    |    x|'
```

String Formatting (3)

- Format specifiers for number formatting (precision, exponent notation, percentage)

```
>>> # fixed point with two decimals
... "{0:.2f}".format(math.pi)
'3.14'

>>> # exponential with two decimals
... # (right-align at min width 10)
... "result:{0:>12.3e}.".format(0.01)
'result:    1.000e-02.'

>>> # Percentage with single decimal
... "{0:.1%}".format(0.1015)
'10.2%'

>>> # binary, octal, hex, character
... {0:b} {0:o} {0:#x} {0:c}".format(123)
'1111011 173 0x7b {'
```

String Encodings

- Strings are just sequences of 8-bit values to the interpreter.

```
>>> [ord(c) for c in "Camelot"]
[67, 97, 109, 101, 108, 111, 116]
```

- If every value stands for one character, there are only 256 possible characters.
- Encoding determines mapping of byte sequence to characters.
- Default encoding in Python 2.x is ASCII (only 127 characters including control characters). (smallest common subset).
- Problem: How to represent languages other than English?

Strings and Unicode in Python 3

- In Python 3 strings are Unicode by default!
- Unicode covers most of the world's writing systems. 16 bit per char.
- Python 3 uses UTF-8 as a default encoding for source.

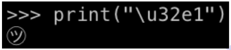
```
>>> x = "smørrebrød"
>>> x
"smørrebrød"
>>> type(x)
class 'str'
>>> len(x)
10
>>> [ord(i) for i in x]
[115, 109, 8960, 114, 114, 101, 98, 114, 8960,
    100]
```

Strings and Unicode in Python 3

- In Python 3 strings are Unicode by default!
- Unicode covers most of the world's writing systems. 16 bit per char.
- Python 3 uses UTF-8 as a default encoding for source.

```
>>> x = "smørrebrød"
>>> x
"smørrebrød"
>>> type(x)
class 'str'
>>> len(x)
10
>>> [ord(i) for i in x]
[115, 109, 8960, 114, 114, 101, 98, 114, 8960,
    100]
```

- Can use explicit codepoints:



Byte sequences

- Python 3 provides `bytes` data type to represent sequence of bytes? (8 bit each)
- Useful for file i/o (binary data).

```
>>> x = b"foobar"
>>> x
b"foobar"
>>> type(x)
<class 'bytes'>
```

Decoding and Encoding

- convert Unicode strings into 8-bit strings using encode. Often used for file output.

```
>>> s = "smørrebrød"
>>> s
'sm\xf8rrebr\xf8d'
>>> len(s)
>>> s_enc = s.encode("UTF-8")
>>> s_enc
b'sm\xc3\xb8rrebr\xc3\xb8d'
>>> len(s_enc)
12
```

- decode 8-bit strings into Unicode strings. Often used for file input.

```
>>> s_enc.decode("UTF-8")
"smørrebrød"
```

- Other encodings:
  latin_1, cp1252 (Windows), mac_roman (Mac), big5 (Chinese), ISO-8859-6 (Arabic), ISO 8859-8 (Hebrew) ..

Difference between Python 2 and 3

- In Python 2, strings are byte sequences (encoded characters).
- Default encoding is ASCII (8 bit per character).
- Python 2 has special unicode strings

```
u"I'm a string"
```

- Python 2 does not have the `byte` datatype.
- Can use `u` prefix in Python 3 to maintain downward compatibility.

Strings

Files and IO

Functions

Simple Input

- Python 3:
  - `input([prompt_str])` writes `prompt_str`, then waits for user to type in a string and press return.
  - Returns a unicode string.
- Python 2.x:
  - `raw_input([prompt_str])` reads in a string (encoded sequence of bytes).
  - `input([prompt_str])` writes `prompt_str`, then reads in a string and evaluates **it as a Python expression.**

```
$python2.7
>>> x = input("list? ")
list? [1,2,3]
>>> type(x)
<type 'list'>
>>> input() # Can be dangerous
sys.exit(1)
```

File Objects

- To read or write a file it has to be opened.
- `open(filenam_str, [mode], [encoding=encoding])` returns an object of class `_io.TextIOWrapper`.
- mode is a string determining operations permitted on the file object.
  - 'r': read only, 'w': write only, 'a': append at the end.
- encoding is an encoding.

```
>>> f = open('testfile.test','w', encoding="ASCII")
>>> f
<_io.TextIOWrapper name='testfile.test' mode='w'
    encoding='UTF-8'>
>>> f.close()
>>> open("test.text","rb")
>>> <_io.BufferedReader name='test.text'>
```

Files and Encodings

- can add keyword parameter to open to specify encoding (default: UTF-8).
- appending 'b' to the mode opens file it in binary mode. (encoding doesn't make sense then)
- Reading from binary file produces `byte` objects.

```
>>> f= open('testfile.test','w', encoding="ASCII")
>>> f
<_io.TextIOWrapper name='testfile.test' mode='w'
    encoding='ASCII'>
>>> f.close()
>>> f = open("test.text","rb")
>>> f
<_io.BufferedReader name='test.text'>
```

Reading from Text Files - Linewise Reading

File nee.txt:

```
ARTHUR: Who are you?
KNIGHT: We are the Knights Who Say... Nee!
```

- Return a single line every time `file.readline()` is called (including `\n`).
- readline() Returns an empty string if there is no more line.

```
>>> f = open('nee.txt','r')
>>> l = f.readline()
>>> while l:
...     print(l)
...     l = f.readline()
...
ARTHUR: Who are you?

KNIGHT: We are the Knights Who Say... Nee!
```

Reading from Text Files - Textfiles as iterators

File nee.txt:

```
ARTHUR: Who are you?
KNIGHT: We are the Knights Who Say... Nee!
```

- Can use file objects as an iterator.

```
>>> f = open('nee.txt','r')
>>> for l in f:
...    print(l)
...
ARTHUR: Who are you?

KNIGHT: We are the Knights Who Say... Nee!
```

Reading from Text Files - readlines

File nee.txt:

```
ARTHUR: Who are you?
KNIGHT: We are the Knights Who Say... Nee!
```

- `f.readlines()` returns a list of all lines.

```
>>> f = open('nee.txt','r')
>>> f.readlines()
['ARTHUR: Who are you?\n',
 'KNIGHT: We are the Knights Who Say... Nee!\n']
```

Reading from Files - read() and seek()

- `f.read([size])` reads (at most) the next `size` characters.
  - if `size` is not specified, the whole file is read.
  - returns empty string if no more bytes available.
- `f.seek(offset)` jumps to position `offset` in the file.

File test.txt:

```
This is a test.
```

```
>>> f = open("test.txt","r")
>>> f.read()
'This is a test file. \n'
>>> f.seek(0)
>>> s = f.read(10)
>>> while s:
...     print s
...     s = f.read(10)
...
This is a
test.
```

Writing to Files

- `f.write(str)` writes `str` to the file.
- `f.writelines(iter)` writes each string from an iterator to a file, adding linebreaks.
- Need to close file with `f.close()` to ensure everything is written from the internal buffer.
- Can also use `f.flush()` to force writeback without closing.

```
>>> f = open("test2.txt","w")
>>> f.write("hello! ")
>>> f.writelines(["a","b","c"])
>>> f.close()
```

test2.txt:

```
hello! a
b
c
```

stdin and stdout

- Can access terminal input (`sys.stdin`) and terminal output (`sys.stdout`) as a file object.
- These objects are defined globally in the module `sys`.

```
>>> import sys
>>> sys.stdout.write("Hello world!\n")
Hello world!
>>> sys.stdin.read(4);
23423
'2342'
```

Strings

Files and IO

Functions

Functions

- Subroutine that compute some result, given its parameters.

```
def pythagoras(leg_a,leg_b):
    """ Compute the length of the hypotenuse
    opposite of the right angle between leg_a
    and leg_b.
    """
    hypotenuse = math.sqrt(leg_a**2 + leg_b**2)
    return hypotenuse
```

```
>>> pythagoras(3.0, 4.0) # Function call passes
   arguments
5.0
```

- More readable code: Break up code into meaningful units.
- Avoid duplicate code.
- Can be shared through modules.
- Abstract away from concrete problem.
- Powerful computational device: allow recursion.

Function definitions

```
def function_name(parameter_1, ..., parameter_n):
    """
    A docstring describing the function.
    """
    statements
    ...
    return result
```

- convention for function names and formal parameters: `lower_case_with_underscore`
- Docstring, parameters, and `return` are optional.
- `return` can occur anywhere in the function.
  - terminates the function and returns the return value (or `None` if no value is provided)
  - A function with no return statement returns `None` once if there are no more statements to execute.

Function Calls

- When a function is called, arguments are passed through its formal parameters.
- The parameter names are used as variables inside the function.
- Python uses call by object: parameters are names for objects.

```
foo(arg1, arg2)
foo  # Not a function call (see later)
```

Parameters with Default Value

- Function definition can assign default value to parameters.
- When no argument is passed during a function call, default value is assumed.
- Default values are computed when function is defined!

```
>>> def test(a, b=[1,2,3]):
...     b.append(a)
...     return b
...
>>> test(1)
[1, 2, 3, 1]
>>> # Watch out for mutable objects in default
    parameters
... test(2)
[1, 2, 3, 1, 2]
```

Extra Positional and Named Arguments

- *args defines an arbitrary list of additional positional arguments.

```
>>> def foo(*numbers):
...     print(type(numbers))
...     print(len(numbers))
...     return sum(numbers)
...
>>> foo(1,2,3)
<type 'tuple'>
3
6
```

Scope

- A function's parameters and any variables defined in the function are in local scope.
- These variables are not visible in surrounding scopes.
- Variables defined in surrounding scope are visible.
  - re-assigning them creates a new local variable!
- Scope is determined statically, variable binding dynamically.
- Loops do not define local scope in Python.

```
a = 1

def foo(b):
    c = 2
    # a is the surrounding a

def bar(b): #different b
    c = 3   #different c
    a = 3   # Create new local variable a

# cannot see either b or c
```

Functions as first-order objects

- First-order objects:
  - anything that can be
    - assigned to a variable
    - stored in a collection
    - passed as a parameter
    - returned by a function
  - In Python pretty much anything is a first-order object, including functions.

```
def add(a, b):
    return a + b

def mult(a, b):
    return a * b

def apply(fun, a, b):
    return fun(a, b)

print(apply(add, 2, 3)) # 5
print(apply(mult, 2, 3)) # 6
```

Functions and Iterators: Map, Filter

- map: return a list containing the result of some function applied to each object in a collection.

```
>>> def quadruple(x):
...     return x ** 4
...
>>> x = map(quadruple, range(5))
>>> x
>>> <map object at 0x10f76f358>
>>> list(x)
[0, 1, 16, 81, 256]
```

- filter: retain only elements for which the function returns `True`.

```
>>> def is_even(x):
...     return x % 2 == 0
...
>>> list(filter(is_even, range(11)))
[0, 2, 4, 6, 8, 10]
```

Anonymous Functions, lambda Expressions

- Defining functions with def can be verbose.
- Want to define small function objects in-place.
  - map, filter, sort.
- `lambda argument1, ... : expression`

```
>>> x = filter(lambda x: x % 2 == 0, range(11))
>>> list(x)
[0, 2, 4, 6, 8, 10]
```

Another Example: Sorting Complex Objects

```
>>> x = [(1,'b'),(4,'a'),(3,'c')]
>>> x.sort()
>>> x
[(1, 'b'), (3, 'c'), (4, 'a')]
```

- Can use function objects to sort by second element.

```
>>> x.sort(key = lambda item: item[1])
[(4, 'a'), (1, 'b'), (3, 'c')]
```

- (better to use `itemgetter`)

```
>>> from operator import itemgetter
>>> x.sort(key = itemgetter(1))
```

Recursion

- Functions can call themselves in their definition.
  - Creates a looping behavior.
  - Divides problems into sub-problems.
- Intuitive way to describe some algorithms.

```
def fac(n):
    """ Compute n!
    """
    if n == 0:    # base case.
        return 1
    else:
        return n * fac(n-1)
```

Generators

- Often a function needs to produce a number of values (a sequence).
- Each result returns only on previous results.
- Storing the whole sequence is memory intensive.
- Generator:
  - An iterator that compute it's next element 'lazily' (on-demand).
  - Can be defined by using the keyword `yield` within a function.
  - Function is executed up to `yield` and interrupted

```
>>> def fib():
...     a, b = 0, 1
...     while True:
...         yield a
...         a, b = b, a + b
>>> fib()
<generator object fib at 0x10c1d60a0>
```

A Generator for the Fibonacci Sequence

```
>>> def fib():
...     a, b = 0, 1
...     while True:
...         yield a
...         a, b = b, a + b
>>> fib()
<generator object fib at 0x10c1d60a0>
>>> for num in fib(): # infinite loop
...     print num
1
1
2
3
5
...
```

Scope

- A function's parameters and any variables defined in the function are in its local scope.
- These variables are not visible in surrounding scopes.
- Names defined in surrounding scope are visible in the function.
  - They point to the object bound to them when function is called.
  - Re-assigning them creates a new local variable!

```
a = 1

def foo(b):
    c = 2
    # a is the surrounding a

def bar(b): #different b
    c = 3   #different c
    a = 3   # Create new local variable a

# cannot see either b or c
```

Scope revisited (1) - What does this program print?

```
x = 3

def foo():
    print(x)

x = 2

def spam(x):
    print(x)

def bar():
    x = 7
    print(x)

def eggs():
    print(x)
    x = 5
```

```
print(x)

foo()

spam(9)

print(x)

bar()

eggs()
```

Scope revisited (1) - What does this program print?

```
x = 3

def foo():
    print(x)

x = 2

def spam(x):
    print(x)

def bar():
    x = 7
    print(x)

def eggs():
    print(x)
    x = 5
```

```
print(x)

foo()

spam(9)

print(x)

bar()

eggs()
```

- 2

Scope revisited (1) - What does this program print?

```
x = 3

def foo():
    print(x)

x = 2

def spam(x):
    print(x)

def bar():
    x = 7
    print(x)

def eggs():
    print(x)
    x = 5
```

```
print(x)

foo()

spam(9)

print(x)

bar()

eggs()
```

- 2
- 2

Scope revisited (1) - What does this program print?

```
x = 3

def foo():
    print(x)

x = 2

def spam(x):
    print(x)

def bar():
    x = 7
    print(x)

def eggs():
    print(x)
    x = 5
```

```
print(x)

foo()

spam(9)

print(x)

bar()

eggs()
```

- 2
- 2
- 9

Scope revisited (1) - What does this program print?

```
x = 3

def foo():
    print(x)

x = 2

def spam(x):
    print(x)

def bar():
    x = 7
    print(x)

def eggs():
    print(x)
    x = 5
```

```
print(x)

foo()

spam(9)

print(x)

bar()

eggs()
```

- 2
- 2
- 9
- 2

Scope revisited (1) - What does this program print?

```
x = 3

def foo():
    print(x)

x = 2

def spam(x):
    print(x)

def bar():
    x = 7
    print(x)

def eggs():
    print(x)
    x = 5
```

```
print(x)

foo()

spam(9)

print(x)

bar()

eggs()
```

- 2
- 2
- 9
- 2
- 7

Scope revisited (1) - What does this program print?

```
x = 3

def foo():
    print(x)

x = 2

def spam(x):
    print(x)

def bar():
    x = 7
    print(x)

def eggs():
    print(x)
    x = 5
```

```
print(x)

foo()

spam(9)

print(x)

bar()

eggs()
```

- 2
- 2
- 9
- 2
- 7

- NameError: name 'x' is not defined

Scope is determined statically, variable bindings are determined dynamically.

Scope revisited (2) - What does this program print?

```
x = 3

print(x)

for x in range(2):
    print(x)

print(x)
```

Scope revisited (2) - What does this program print?

```
x = 3

print(x)

for x in range(2):
    print(x)

print(x)
```

- 3
- 0
- 1

Scope revisited (2) - What does this program print?

```
x = 3

print(x)

for x in range(2):
    print(x)

print(x)
```

- 3
- 0
- 1
- 1

- Block structure (specifically loops) does not define scope!

Nested Functions

```
>>> def a():
...     print('spam')
...
>>> def b():
...     def a():
...         print('eggs')
...     a()
...
>>> a()
spam
>>> b()
eggs
>>> a()
spam
```

- Function definitions can be nested.
- Function names are just variables bound to function objects (first-class functions).
- Therefore the same scoping rules as for variables apply.

Closures

- Nested functions can be used to create closures.
- Closure: Function object that contains some 'state'.
  - Function refers to variables that are bound outside its local scope when function object is created.

```
>>> def make_power_func(x):
...     def power(y):
...         return y**x
...     return power
...
>>> power_two = make_power_func(2)
>>> power_two(4)
16
```

x is in the surrounding scope of `power`. Its binding is preserved when `power` is defined (i.e. when `make_power_func` is called).