Modules

Modules

- collection of functions and variables, typically in scripts
  - file name: *module name* + '.py'
- Example: fib, fib2 functions defined in file 'fibo.py'
- executed only when module is imported

```
>>> import fibo
```

- Use modules via "name space":

```
>>> fibo.fib(1000)
>>> fibo.__name__
```

- can give it a local name:

```
>>> fib = fibo.fib
>>> fib(500)
```

- can import into name space:

```
>>> from fibo import fib, fib2
>>> fib(500)
```

- can import all names defined by module:

```
>>> from fibo import *
```

Module search path

- current directory
- list of directories specified in PYTHONPATH environment variable
- uses installation-default if not defined, e.g.) /usr/local/lib/python
- uses sys.path (`sys` module: system dependent list)

```
>>> import sys
>>> sys.path
['', 'C:\\Python27\\Lib\\idlelib', 'C:\\WINDOWS\\SYSTEM32\\python27.zip', 'C:\\P
ython27\\DLLs', 'C:\\Python27\\lib', 'C:\\Python27\\lib\\plat-win', 'C:\\Python2
7\\lib\\lib-tk', 'C:\\Python27', 'C:\\Python27\\lib\\site-packages']
```

Compiled Python files

- include byte-compiled version of module if there exists fibo.pyc in same directory as fibo.py
  - only if creation time of fibo.pyc matches fibo.py
- automatically write compiled file, if possible
- platform independent
- doesn't run any faster, but *loads* faster
- can have only .pyc file → hide source

Command-Line Arguments

- sys.argv

Java: `public static void main (String argv[])`

C: `void main (int argc, char **argv)`

`test.py`

```
import sys

for arg in sys.argv:
    print arg
```

`argv[0]` is always the program itself
(like C, but unlike Java)

```
$ python test.py
test.py
$ python test.py abc def
test.py
abc
def
$ python test.py --help
test.py
--help
$ python test.py -m kant.xml
test.py
-m
kant.xml
```

Input and Output

Example: Word frequency - 1

```
import sys
freq = {}   # frequency of words in text

for line in sys.stdin:
    for word in line.split():
        freq[word] = freq.get(word,0)

for w in sorted(freq.keys()):
    print w, freq[w]
```

Example: Word Frequency - 2

```
import sys
from operator import itemgetter

punctuation = """'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'"""

freq = {}    # frequency of words in text

stop_words = {}
for line in open("stop_words.txt"):
    stop_words[line.strip()] = True

for line in sys.stdin:
  for word in line.split():
    word = word.strip(punctuation).lower()
      if not word in stop_words:
        freq[word] = freq.get(word,0) + 1

words = sorted(freq.iteritems(), key=itemgetter(1), reverse=True)

for w,f in words:
    print w, f
```

Common File Operations

Operation	Interpretation
`output = open(r'C:\spam', 'w')`	Create output file (`'w'` means write)
`input = open('data', 'r')`	Create input file (`'r'` means read)
`input = open('data')`	Same as prior line (`'r'` is the default)
`aString = input.read()`	Read entire file into a single string
`aString = input.read(N)`	Read up to next N characters (or bytes) into a string
`aString = input.readline()`	Read next line (including `\n` newline) into a string
`aList = input.readlines()`	Read entire file into list of line strings (with `\n`)
`output.write(aString)`	Write a string of characters (or bytes) into file
`output.writelines(aList)`	Write all line strings in a list into file
`output.close()`	Manual close (done for you when file is collected)
`output.flush()`	Flush output buffer to disk without closing
`anyFile.seek(N)`	Change file position to offset N for next operation
`for line in open('data'): use line`	File iterators read line by line
`open('f.txt', encoding='latin-1')`	Python 3.X Unicode text files (`str` strings)
`open('f.bin', 'rb')`	Python 3.X bytes files (`bytes` strings)
`codecs.open('f.txt', encoding='utf8')`	Python 2.X Unicode text files (`unicode` strings)
`open('f.bin', 'rb')`	Python 2.X bytes files (`str` strings)

File Objects

- Files
  - text files: always contain strings: `str`
  - binary files: contain raw 8-bit bytes: `bytes` (e.g. `b'hello world!'`)
- Opening/closing files
  - file_obj = `open`(*file_name*, *[mode]*)
    - mode
      - `'r'`: for reading (default)
      - `'w'`: for writing (truncate if already exists)
      - `'a'`: for appending
      - `'r+'`: for reading and writing
      - `'w+'`: for reading and writing (truncate if already exists)
      - `'b'`: binary file
      - `'U'`: universal newlines (`'\n'`, `'\r\n'`, `'\r'`를 지원)
  - file_obj.`close`()
    - Python session이 끝날 때, interpreter가 garbage collection할 때, 자동으로 close
- File object is an **iterator**
- `sys.stdin, sys.stdout, sys.error` file object인 interpreter 시작시 open됨

Examples: File Operations

```
>>> myfile = open('myfile.txt', 'w')          # Open for text output: create/empty
>>> myfile.write('hello text file\n')         # Write a line of text: string
16
>>> myfile.write('goodbye text file\n')
18
>>> myfile.close()                            # Flush output buffers to disk

>>> myfile = open('myfile.txt')               # Open for text input: 'r' is default
>>> myfile.readline()                         # Read the lines back
'hello text file\n'
>>> myfile.readline()
'goodbye text file\n'
>>> myfile.readline()                         # Empty string: end-of-file
''
```

```
>>> open('myfile.txt').read()                 # Read all at once into string
'hello text file\ngoodbye text file\n'

>>> print(open('myfile.txt').read())          # User-friendly display
hello text file
goodbye text file
```

Iterating over lines

- Old style

```
file = open('data.txt')
for line in file.readlines():
    print line,
```

- File is iterable!!

```
file = open('data.txt')
for line in file:
    print line,
```

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

- New – auto closing and exception handling

```
with open("data.txt") as f:
    for line in f:
        print line
```

Iterators and list

```
>>> open('data.txt').readlines()                          # always read lines
['Hello file world!\n', 'Bye   file world.\n']

>>> list(open('data.txt'))                                # force line iteratior
['Hello file world!\n', 'Bye   file world.\n']

>>> lines = [line.rstrip() for line in open('data.txt')]  # comprehension
>>> lines
['Hello file world!', 'Bye   file world.']

>>> lines = [line.upper() for line in open('data.txt')]   # arbitrary actions
>>> lines
['HELLO FILE WORLD!\n', 'BYE   FILE WORLD.\n']

>>> list(map(str.split, open('data.txt')))                # apply a function
[['Hello', 'file', 'world!'], ['Bye', 'file', 'world.']]

>>> line = 'Hello file world!\n'
>>> line in open('data.txt')                              # line membership
True
```

Text and Binary Files

- Text files represent content as normal `str` strings,
  - perform end-of-line translation by default.
- Binary files represent content as a special `bytes` string type
  - allow programs to access file content unaltered.

```
>>> data = open('data.bin', 'rb').read()      # Open binary file: rb=read binary
>>> data                                      # bytes string holds binary data
b'\x00\x00\x00\x07spam\x00\x08'
>>> data[4:8]                                 # Act like strings
b'spam'
>>> data[4:8][0]                              # But really are small 8 bit integers
115
```

Storing Python Objects in Files: Conversions

- must convert objects to strings using conversion tools

```
>>> X, Y, Z = 43, 44, 45                    # Native Python objects
>>> S = 'Spam'                              # Must be strings to store in file
>>> D = {'a': 1, 'b': 2}
>>> L = [1, 2, 3]
>>>
>>> F = open('datafile.txt', 'w')           # Create output text file
>>> F.write(S + '\n')                       # Terminate lines with \n
>>> F.write('%s,%s,%s\n' % (X, Y, Z))       # Convert numbers to strings
>>> F.write(str(L) + '$' + str(D) + '\n')   # Convert and separate with $
>>> F.close()
```

```
>>> chars = open('datafile.txt').read()     # Raw string display
>>> chars
"Spam\n43,44,45\n[1, 2, 3]${'a': 1, 'b': 2}\n"
>>> print(chars)                            # User-friendly display
Spam
43,44,45
[1, 2, 3]${'a': 1, 'b': 2}
```

```
>>> F = open('datafile.txt')                # Open again
>>> line = F.readline()                     # Read one line
>>> line
'Spam\n'
>>> line.rstrip()                           # Remove end-of-line
'Spam'
```

Storing Native Python Objects: pickle

- to store almost any Python object in a file directly

```
>>> D = {'a': 1, 'b': 2}
>>> F = open('datafile.pkl', 'wb')
>>> import pickle
>>> pickle.dump(D, F)                    # Pickle any object to file
>>> F.close()
```

```
>>> F = open('datafile.pkl', 'rb')
>>> E = pickle.load(F)                   # Load any object from file
>>> E
{'a': 1, 'b': 2}
```

```
>>> open('datafile.pkl', 'rb').read()          # Format is prone to change!
b'\x80\x03}q\x00(X\x01\x00\x00\x00bq\x01K\x02X\x01\x00\x00\x00aq\x02K\x01u.'
```

Storing Python Objects in JSON Forma

- JSON is a newer and emerging data interchange format
  - programming-language-neutral
  - does not support as broad a range of Python object types as pickle

```
>>> name = dict(first='Bob', last='Smith')
>>> rec  = dict(name=name, job=['dev', 'mgr'], age=40.5)
>>> rec
{'job': ['dev', 'mgr'], 'name': {'last': 'Smith', 'first': 'Bob'}, 'age': 40.5}
```

```
>>> import json
>>> json.dumps(rec)
'{"job": ["dev", "mgr"], "name": {"last": "Smith", "first": "Bob"}, "age": 40.5}'

>>> S = json.dumps(rec)
>>> S
'{"job": ["dev", "mgr"], "name": {"last": "Smith", "first": "Bob"}, "age": 40.5}'

>>> O = json.loads(S)
>>> O
{'job': ['dev', 'mgr'], 'name': {'last': 'Smith', 'first': 'Bob'}, 'age': 40.5}
>>> O == rec
True
```

JSON

```
>>> json.dump(rec, fp=open('testjson.txt', 'w'), indent=4)
>>> print(open('testjson.txt').read())
{
    "job": [
        "dev",
        "mgr"
    ],
    "name": {
        "last": "Smith",
        "first": "Bob"
    },
    "age": 40.5
}
>>> P = json.load(open('testjson.txt'))
>>> P
{'job': ['dev', 'mgr'], 'name': {'last': 'Smith', 'first': 'Bob'}, 'age': 40.5}
```

Shelve – persistent dictionary-like obj

```
from initdata import bob, sue
import shelve
db = shelve.open('people-shelve')
db['bob'] = bob
db['sue'] = sue
db.close()
```

```
from initdata import tom
import shelve
db = shelve.open('people-shelve')
sue = db['sue']
sue['pay'] *= 1.50
db['sue'] = sue
db['tom'] = tom
db.close()
```

- supports most of the same functionality as dictionaries
- modified objects are written *only* when assigned to the shelf
- the values (not the keys!) in a shelf can be essentially arbitrary Python objects

flag	Meaning
'r'	Open existing database for reading only
'w'	Open existing database for reading and writing
'c'	Open database for reading and writing, creating it if it doesn't exist (default)
'n'	Always create a new, empty database, open for reading and writing

Iterators and Generators

Iterators

- `for` statement calls `iter()` on the container object
- `iter()` returns an iterator object that defines the method `next()` (`__next__()` in Python 3)

```
>>> for char in 'abc':
        print char,


a b c
```

```
>>> it = iter('abc')
>>> it
<iterator object at 0x00000000032657F0>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()

Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    it.next()
StopIteration
```

```
>>> for line in open('path.py'):
        print line,


import sys
print sys.path
```

```
>>> f = open('path.py')
>>> f.next()
'import sys\n'
>>> f.next()
'print sys.path\n'
>>> f.next()

Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    f.next()
StopIteration
```

Generators

- A simple and powerful tool for creating iterators
  - like regular functions but use the `yield` statement whenever they want to return data.
  - Each time `next()` is called on it, the generator resumes where it left off

```
>>> def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]


>>> for char in reverse('golf'):
        print char


f
l
o
g
```

Generator Expressions

- Generator expressions are more compact but less versatile than full generator definitions
- more memory friendly than equivalent list comprehensions

```
>>> (i*i for i in range(10))
<generator object <genexpr> at 0x0000000002DE0FC0>
>>> list(i*i for i in range(10))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [i*i for i in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> sum(i*i for i in range(10))
285
```

Exception Handling

C Program

```
doStuff()
{                                       # C program
    if (doFirstThing() == ERROR)        # Detect errors everywhere
        return ERROR;                   # even if not handled here
    if (doNextThing() == ERROR)
        return ERROR;
    ...
    return doLastThing();
}

main()
{
    if (doStuff() == ERROR)
        badEnding();
    else
        goodEnding();
}
```

Python Program

```
def doStuff():          # Python code
    doFirstThing()      # We don't care about exceptions here,
    doNextThing()       # so we don't need to detect them
    ...
    doLastThing()

if __name__ == '__main__':
    try:
        doStuff()       # This is where we care about results,
    except:             # so it's the only place we must check
        badEnding()
    else:
        goodEnding()
```

Exceptions

- Python raises exceptions whenever it detects errors
  - default exception-handling: stops the program, or
  - use a `try` statement to catch and recover from the exception

```
>>> def fetcher(obj, index):
        return obj[index]

>>> x = 'Spam'
```

raise IndexError

default exception handler

```
>>> fetcher(x, 4)

Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    fetcher(x, 4)
  File "<pyshell#20>", line 2, in fetcher
    return obj[index]
IndexError: string index out of range
```

your exception handler

```
>>> try:
        fetcher(x, 4)
except IndexError as e:
        print 'Got exception:', e


Got exception: string index out of range
```

Raising exceptions

```
>>> try:
...     raise IndexError
... except IndexError:
...     print('got exception')
...
got exception
```

exception is a class object!

assert:
conditional raise

```
>>> assert False, 'Nobody expects the Spanish Inquisition!'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Nobody expects the Spanish Inquisition!
```

user-defined exception

```
>>> class AlreadyGotOne(Exception): pass      # User-defined exception

>>> def grail():
        raise AlreadyGotOne()                 # Raise an instance

>>> try:
...     grail()
... except AlreadyGotOne:                     # Catch class name
...     print('got exception')
...
got exception
```

Handling Exception

May raise exception

```
f = open("myfile.txt")
for line in f:
    print line,
```

Exception handling

```
try:
    f = open("myfile.txt")
except IOError as e:  # exceptions occur
    print 'cannot open:', e
else:                 # no exceptions
    for line in f:
        print line,
finally:
    f.close()
```

Pre-defined Clean-up

file automatically closed on exit

```
with open("myfile.txt") as f:
    for line in f:
        print line,
```

Raising Exceptions
User-defined Exceptions

```
>>> try:
...       raise Exception('spam', 'eggs')
... except Exception as inst:
...       print type(inst) # the exception instance
...       print inst.args # arguments stored
in .args
...       print inst # __str__ allows args to be
printed directly
...       x, y = inst.args
...       print 'x =', x
...       print 'y =', y
...
<type 'exceptions.Exception'>
('spam', 'eggs') ('spam', 'eggs')
x = spam
y = eggs
```

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print 'My exception occurred,
value:', e.value
...
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

Exception Hierarchy

See Exception Hierarchy

Clause form	Interpretation
except:	Catch all (or all other) exception types.
except *name*:	Catch a specific exception only.
except *name* as *value*:	Catch the listed exception and assign its instance.
except (*name1*, *name2*):	Catch any of the listed exceptions.
except (*name1*, *name2*) as *value*:	Catch any listed exception and assign its instance.
else:	Run if no exceptions are raised in the `try` block.
finally:	Always perform this block on exit.

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
      |              +-- UnicodeDecodeError
      |              +-- UnicodeEncodeError
      |              +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
```

Functional Programming

Functional programming tools

- intuition: function as data
- filter(function, sequence)
  - pass items of the sequence only if function(item) == True

```
>>> def f(x): return x%2 != 0 and x%3 == 0
>>> filter(f, range(2,25))
[3, 5, 6, 9, 10, 12, 15, 18, 20, 21, 24]
```

- map(function, sequence)
  - call function for each item
  - return `list` of return values

```
>>> seq = range(8)
>>> def add(x, y): return x+y

>>> map(add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

- reduce(function, sequence)
  - return a single value
  - call binary function on the first two items
  - then on the result and next item

lambda

- map/filter in one line for custom functions?
  - “anonymous inline function”
- borrowed from LISP, Scheme, ML, OCaml

```
>>> f = lambda x: x*2
>>> f(1)
2
>>> map (lambda x: x**2, [1, 2])
[1, 4]
>>> filter (lambda x: x > 0, [-1, 1])
[1]
>>> g = lambda x,y : x+y
>>> g(5,6)
11
>>> map (lambda (x,y): x+y, [(1,2), (3,4)])
[3, 7]
```

more on lambda

```
>>> f = lambda : "good!"
>>> f
<function <lambda> at 0x381730>
>>> f()
'good!'
```

lazy evaluation

```
>>> a = [5, 1, 2, 6, 4]
>>> a.sort(lambda x,y : y - x)
>>> a
[6, 5, 4, 2, 1]
```

custom comparison

map, filter, and list comprehension

- `map(f, a) ≡ [f(x) for x in a]`
- `filter(p, a) ≡ [x for x in a if p(x)]`
- `map(f, filter(p, a)) ≡ [f(x) for x in a if p(x)]`

```
>>> map(int, ('1', '2'))
[1, 2]
>>>
>>> map(int, ('1', '2'))
[1, 2]
>>> " ".join(map(str, ['1', '2']))
'1 2'
```

```
>>> def is_even(x): return x % 2 == 0

>>> filter(is_even, [-1, 0])
[0]
>>> filter(is_even, [-1, 0, 1, 2])
[0, 2]
>>> filter(lambda x: x %2 == 0, [-1, 0, 1, 2])
[0, 2]
>>> [x for x in [-1, 0, 1, 2] if is_even(x)]
[0, 2]
>>> [x for x in [-1, 0, 1, 2] if lambda x: x % 2 == 0]
[-1, 0, 1, 2]
```

Exercises

- 1. Write a program which can filter even numbers in a list by using filter function. The list is: [1,2,3,4,5,6,7,8,9,10].
  - Hints:
    - Use filter() to filter some elements in a list.
    - Use lambda to define anonymous functions.
- 2. Write a program which can map() to make a list whose elements are square of elements in [1,2,3,4,5,6,7,8,9,10].
  - Hints:
    - Use map() to generate a list.
    - Use lambda to define anonymous functions.
- 3. Write a program which can map() and filter() to make a list whose elements are square of even number in [1,2,3,4,5,6,7,8,9,10].
  - Hints:
    - Use map() to generate a list.
    - Use filter() to filter elements of a list.
    - Use lambda to define anonymous functions.

reduce

- apply binary operator recursively

(((((1+2)+3)+4)+5)

```
>>> reduce(lambda x,y : x*y, [1,2,3,4,5])
120
>>> reduce(lambda x,y : x+y, [1,2,3,4,5])
15
>>> reduce(lambda x,y : x+y, [2])
2
>>> reduce(lambda x,y : x+y, [])
TypeError: reduce() of empty sequence with no initial value
```

- return initial value if sequence is empty

```
>>> reduce(lambda x,y : x+y, [], 0)
0
```

```
>>> def my_sum(seq):
        return reduce(lambda x,y : x+y, seq, 0)

>>> my_sum([1,2,3,4,5])
15
>>> my_sum([])
0
```

implementing reduce()

```
>>> def myreduce(f, seq, initial = None):
...     if seq == []:
...             return initial
...     if len(seq) == 1:
...             return seq[0]
...     return f( myreduce(f,seq[:-1],initial), seq[-1]
...
>>> myreduce(lambda x,y: x+y, [1,2,3])
6

>>> def smaller(a,b):
...     if a < b:
...             return a
...     return b
...
>>> f = lambda seq: reduce (smaller, seq)
>>> f([1,0,2,5,-1])
-1
>>> min([1,0,2,5,-1])
-1
```

min(), max() are builtin

Functional Style

- higher-order functions (taking functions as arguments)
- almost no assignments (no side conditions)
- often recursive, and sometimes lazy

```
def perm(n, m, current = []):
   if len(current) == m:
      for elem in current:
         print elem,
      print
      return 1

   sum = 0
   for i in range(1,n+1):
      if i not in current:
         sum += perm( n, m, current + [i] )
   return sum
```

```
def perm(n, m, current = []):
   if len(current) == m:
      print " ".join( map(str, current) )
      return 1

   return sum([ perm(n, m, current + [i]) \
            for i in range(1, n+1) if i not in current ])

print perm (3, 2)
```

Memoization

```
def fib(n):
    a, b = 0, 1
    for i in range(n-1):
        a, b = b, a+b
    return a
```

version 1 (non-recursive)
fast, but counter-intuitive

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

version 2 (recursive)
intuitive, but ...

Don't you know this is very bad?

How bad it is?

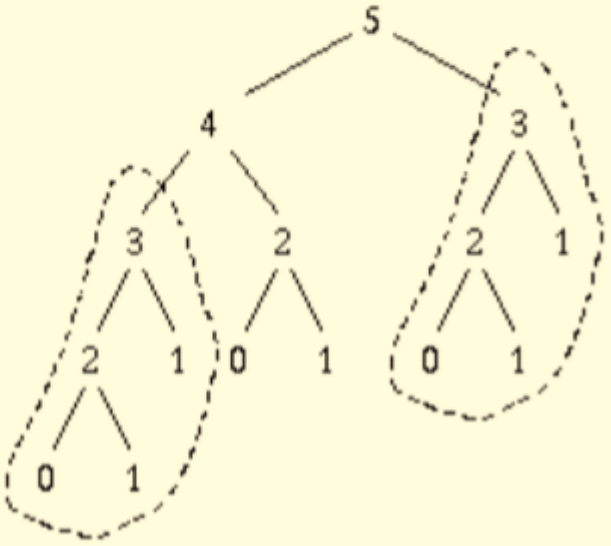
```
def fib(n):
    a, b = 0, 1
    for i in range(n-1):
        a, b = b, a+b
    return a
```

$O(n)$

version I (non-recursive)
fast, but counter-intuitive

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

$O(1.6^n)$



How to Solve this Problem?

- Memoization: Anything recursive can (and should) be memoized to share overlapping subproblems

```
def fib(n):
    a, b = 0, 1
    for i in range(n-1):
        a, b = b, a+b
    return a
```

$O(n)$

version 1 (non-recursive)
fast, but counter-intuitive

```
fibs = {}
def fib(n):
    if n in fibs:
        return fibs[n]
    if n <= 1:
        fibs[n] = n
    else:
        fibs[n] = fib(n-1) + fib(n-2)
    return fibs[n]
```

$O(n)$

version 3 (memoized)
intuitive, and fast!