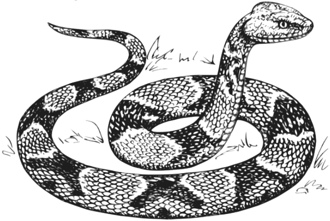
# CS 3101-1 - Programming Languages: Python

## Lecture 1: Introduction / Python Basics

Daniel Bauer (bauer@cs.columbia.edu)

September 10, 2014



# Course Outline

- Lectures:

  Wed 10:10am-12:00pm, 9/10 to 10/15 (6-weeks)
- Instructor:
  - Daniel Bauer (`bauer@cs.columbia.edu`)
  - Office hours: Thu 10:00am-12:00pm, CEPSR/Shapiro 7LW3 (SpeechLab)
- TA:
  - Shubhanshu Yadav (`sy2511@columbia.edu`)
  - Office hours: TBD
- Course website (lecture slides, homework):

  `http://www.cs.columbia.edu/~bauer/cs3101-01`

# Contents

# Syllabus

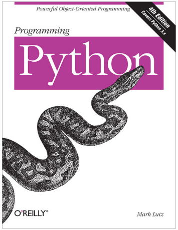
| | |
|---|---|
| Today | Introduction. Python Overview. Basic data types. Control Flow. Tuples. Lists. Dictionaries. Sets. |
| Sep 17 | Strings. Files. I/O. Functions. Lambda expressions. Generators. |
| Sep 24 | More Functions. Object oriented Python: Classes, Objects, Methods, Attributes. Modules and Packages. |
| Oct 1 | Standard Library: sys, os/os.path, collections, pickling. Error Handling, Exceptions. |
| Oct 8 | Regular Expressions. Decorators. Debugging, Unit Testing. |
| Oct 15 | Selected Topics: TBD (numpy/scipy? Web devevopment?) |

# Grading / Deliverables

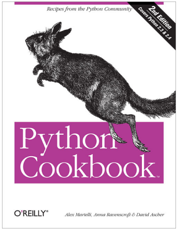
- Class Participation: 10%
- 5 Homeworks: 50%
  - Due following week before class. No late submissions!
  - Small programming tasks.
- Project proposal: 5%
- Final-project: 35 %
  - Dive deeper into specific topics (e.g. third party libraries) that are useful/interesting to you.
  - Collect experiences with real-world Python.

# Textbooks

- No official textbook.
- Some recommendations:




Mark Lutz: Programming Python (standard textbook)

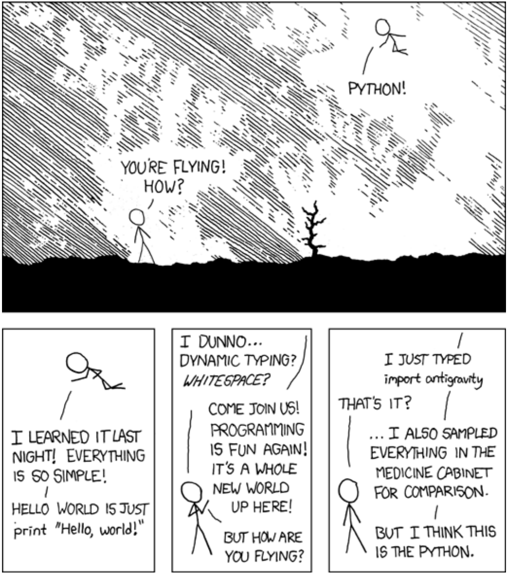



Alex Martelli: Python Cookbook

- More recommendations on course website.

# Online Materials

- Lots of good, up-to-date online material, searchable.
  - Official Python documentation (2.7 and 3)
    `http://docs.python.org/`
  - Official Python tutorial.
    `http://docs.python.org/tutorial/`
  - Online Python Cookbook.
    `http://code.activestate.com/recipes/langs/python/`
  - Mark Pilgrim, Dive into Python 3.
    `http://diveintopython3.ep.io/`




`http://xkcd.org/353/ - import antigravity`

# What is Python

- Versatile, interpreted, high-level, dynamic, programming language.
- Multi paradigm: simple procedural programming, object-orientation and functional programming.
- Popular in science, open source community, web development.
- Scales well to different applications.
- Great developer community.
  - Easy to get help.
  - Lots of available modules.

# Python Design Goals (1)

- Easy to learn and use:
  - automatic memory management.
  - high-level built in data structures.
  - batteries included: Large standard library.
- Readability:
  - intuitive syntax
  - minimal boilerplate.
- Dynamic Behavior:
  - interpreted language.
  - dynamic typing.
  - introspection.

# Python Design Goals (2)

- Portable language:
  - Different interpreters for many platforms: CPython, Jython, IronPython, PyPy.
- Extensibility:
  - Reusable code: Modules and Packages.
  - All of Python is open source.
  - Easy to write new modules in C.

# Python Can Increase Productivity

*"Anecdotal evidence has it that one Python programmer can finish in two months what two C++ programmers can't complete in a year"* [Guido van Rossum, 'Comparing Python to Other Languages', 1997 (online)]

# Python Can Increase Productivity

*" Anecdotal evidence has it that one Python programmer can finish in two months what two C++ programmers can't complete in a year"* [Guido van Rossum, 'Comparing Python to Other Languages', 1997 (online)]

- Interpreted Language: No compilation step.
- Debugging is easy:
  - introspection abilities (state of the interpreter is accessible during program runtime).
  - read-eval-print loop (REPL).
  - elaborate error handling system.
- Faster development times due to more compact code.
  - typically 3-5 times shorter than Java code.
  - typically 5-10 times shorter than C++ code.

# Criticism and Misconceptions

- “Python is a scripting language”
  - False. Python has been used as a scripting language, but it is also used to develop large stand-alone applications.

# Criticism and Misconceptions

- "Python is a scripting language"
  - False. Python has been used as a scripting language, but it is also used to develop large stand-alone applications.
- "Python is interpreted, thus slower than running native code"
  - True, but
    - Most code is not CPU bound, efficiency doesn't matter.
    - Python can be used to 'glue' together native modules.
    - Libraries are often very efficient.

# Criticism and Misconceptions

- "Python is a scripting language"
  - False. Python has been used as a scripting language, but it is also used to develop large stand-alone applications.
- "Python is interpreted, thus slower than running native code"
  - True, but
    - Most code is not CPU bound, efficiency doesn't matter.
    - Python can be used to 'glue' together native modules.
    - Libraries are often very efficient.
- "Whitespaces are ugly."
  - You'll get used to it.

# Criticism and Misconceptions

- "Python is a scripting language"
  - False. Python has been used as a scripting language, but it is also used to develop large stand-alone applications.
- "Python is interpreted, thus slower than running native code"
  - True, but
    - Most code is not CPU bound, efficiency doesn't matter.
    - Python can be used to 'glue' together native modules.
    - Libraries are often very efficient.
- "Whitespaces are ugly."
  - You'll get used to it.
- "Dynamic typing is unsafe."
  - Python is strongly typed and well behaved. It can deal with type errors at runtime.

# Use Cases

- Application development.
  - (from small command line tools, to large GUI based applications and 3D games).
- Web development.
  - (Yelp, YouTube, Reddit, WordsEye, ... )
- Easy to use scripting language.
  - (Emacs, OpenOffice, Blender, various games, ...)
- Scientific/numeric computing.
  - (machine learning, physics, bioinformatics, NLP,...)

# When Not to Use Python

- When implementing low-level routines of CPU bound programs.
- In large multi-threaded applications
  - bad multithreading support.
  - parallelization is well supported.
- In large collaborative projects?
  - Problem of dynamic typing.
  - Needs good documentation / workflow.

# Python Versions

- Two branches:
  - Python 2
    - Current and ultimate release: 2.7
  - Python 3
    - Current release: 3.4.1
    - Some major changes and clean-ups
    - Not backward compatible (cannot execute 2.x code)
    - Ongoing development
- Many important packages not (yet) ported to Python 3.
- *2to3* tool exists, but does not always work correctly.
- This course: Python 3 (some differences to Python 2.7 pointed out).

# Installing Python

- Debian, Ubuntu, etc.

```
$sudo apt-get install python3
```

- OS X MacPorts

```
$sudo port install python34
```

- OS X, Windows
  - Download installer for 3.4.1 at https://www.python.org/downloads/.

# Running Python in Interactive Mode

Python interpreter can be run in an interactive session mode.

- Built-in 'Read/Evaluate/Print-Loop' (REPL)
- Python statements are evaluated and the result is printed to the user.

```
$ python3.4
Python 3.4.1 (default, May 21 2014, 01:39:38)
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang
    -503.0.40)] on darwin
Type "help", "copyright", "credits" or "license"
    for more information.
>>>
>>> print('Hello world!')
Hello world!
>>> 7*6
42
```

- Improved Python shells: IDLE shell, bpython, IPython

# Executing Python Programs on the Command Line

The program ‘hello.py’

```
def main():
    """Greet the user and tell him the answer to life,
       the universe and everything."""

    print('Hello world!') # Be friendly.
    print(7*6)

if __name__=='__main__':
    main()
```

can be run by passing the filename to the python interpreter:

```
$ python hello.py
Hello world!
42
```

# Python IDEs / Editors

IDLE (shipped with Python)
Eclipse (pyDev)
Komodo Edit
Emacs
Vim
Nano
Jedit
Sublime Text
...

- support syntax highlighting, auto-completion
- some support: integrated debugging and profiling.
- this class: enough to use any text editor and command line tools.

# Elementary Python Syntax - Whitespaces and Blocks

## Indentation level and linebreaks are syntactically relevant!

- Single most hated Python feature.
- Actually useful: enforces readable code.

Python

```
while x==1:
....if y:
........f1()
....f2()
```

C/C++/Java...

```
while (x==1) {
    if (y) { f1();}
    f2();
}
```

# Elementary Python Syntax - Whitespaces and Blocks

## Indentation level and linebreaks are syntactically relevant!

- Single most hated Python feature.
- Actually useful: enforces readable code.

Python

```
while x==1:
....if y:
........f1()
....f2()
```

C/C++/Java...

```
while (x==1) {
    if (y) { f1();}
    f2();
}
```

Warning: Never mix tabstops and whitespaces!

- Do not use tabs at all (outside of strings).
- Set your editor/IDE to fill tabs with white spaces automatically.
- Recommendation: 4 spaces per indent level.

# Elementary Python Syntax - Linebreaks

- Compiler ignores blank lines.
- Indentation level only counts after finished lines.
  - if open (, {, or [ has not been closed, the next line is joined automatically.
  - can join lines manually with the \ symbol for readability.
  - sometimes needed with very long lines.

```
# a statement spanning multiple lines
cheeselist = ['cheddar','camembert','swiss',
              'mozarella']

# use \ to join lines
cheeselist = ['cheddar','camembert','swiss', \
             'mozarella']
```

# Elementary Python Syntax - Comments

- Single line comments with # at the end of a line.

```
# Print some informative messages.
print('Hello world!') # Be friendly.
```

# Elementary Python Syntax - Comments

- Single line comments with # at the end of a line.

```
# Print some informative messages.
print('Hello world!') # Be friendly.
```

- 'Docstrings' at the beginning of function, method, class definitions and modules.
  - Are interpreted! Use sparingly!
  - Tripple " surround multi-line strings.
  - Used for documentation (later).

```
def pythagoras(leg_a,leg_b):
    """ Compute the length of the hypotenuse opposite
    of the right angle between leg_a and leg_b."""

    return math.sqrt(leg_a**2 + leg_b**2)
```

# Code Style - Best Practices

- Do not use semicolons ; they are legal, but unnessecary.
- Limit lines to 79 characters.
- Python is case-sensitive:
  - All keywords are lower case.
  - Classnames should be written in CamelCase
  - Everything else (variables, function, modules...) should be `lowercase_with_underscore`.
- Some others (see PEP 8).

# Variables and Assignments

- Evaluate expression on the right hand side of = and assign to it the variable (name) on the left hand side.
- No declaration for variables needed.

```
>>> answer = 6*7
>>> answer
42
>>> answer += 5 # Shortcut += -= *= /=
>>> answer
47
```

- Multiple assignments in one line possible.

```
>>> a, b = 2, 3
>>> a, b = b, a # Swap variables
>>> a
3
>>> b
2
```

# Python Data Types - Built-In Types

- Elementary types
  - NoneType: `None`
  - bool: `True`, `False`
  - (Numeric) int: `42`, long, float: `3.14`, complex: (0.3+2j)
- Container types
  - str: `'Hello'`
  - list: `[1, 2, 3]`
  - tuple: `(1, 2, 3)`
  - dict: `{'A':1, 'B':2}`
  - set: `{1, 2, 3}`
- file
- function, class, instance ...
- In Python *everything* is an object and every object has a type.

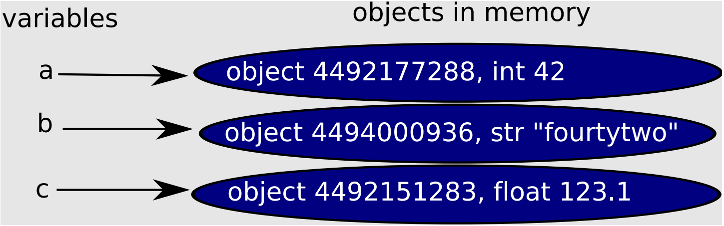
# Python uses Dynamic Typing

- Type checks (making sure variables have the correct type for an operation) performed at runtime.
- No need to declare variable types.
- Can get type of an object with ‘type(variable)’

```
>>> answer = 6*7
>>> answer = 'fortytwo' # create a new string object
...                      # and let answer point to it.
>>> answer
'fortytwo'
>>> type(answer)
<type 'str'>
```

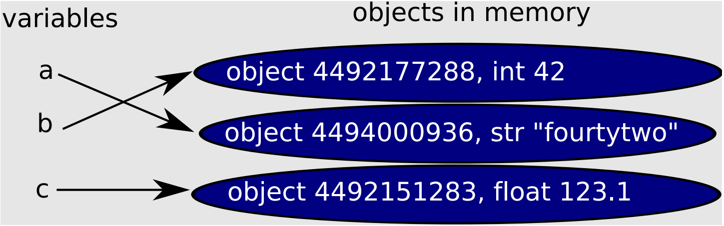
# Variables are Names

**Objects never change their types**, but variables can be names for different objects during runtime.



# Variables are Names

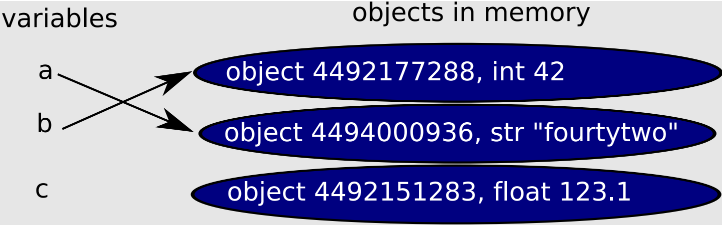
**Objects never change their types**, but variables can be names for different objects during runtime.




```
>>> a,b = b,a
```

# Variables are Names

**Objects never change their types**, but variables can be names for different objects during runtime.

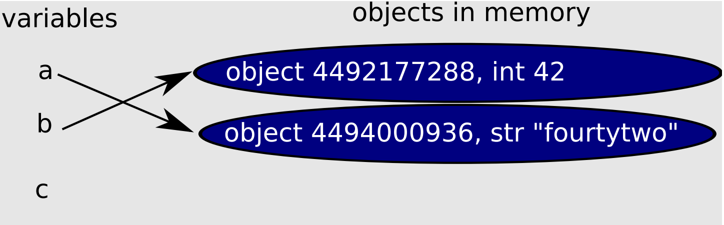



```
>>> a,b = b,a
```

```
>>> c=None
```

# Variables are Names

**Objects never change their types**, but variables can be names for different objects during runtime.




```
>>> a,b = b,a
```

```
>>> c=None
```

Garbage collector removes unreferenced object.

# Python uses Strong Typing

- Operations may expect operands of certain types.
- Interpreter throws an exception if type is invalid.

```
>>> a = 1
>>> b = 'Python'
>>> a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int'
    and 'str'
```

# Mutability

Python has mutable and immutable objects, based on data type.

- Mutable objects (lists, dictionaries, sets) can be modified.

```
>>> cats = ['felix','dinah', 'lucky', 'spot']
>>> cats.append('garfield') # Add an element to
>>>                         # the list object
>>> cats
['felix','dinah','lucky','spot','garfield']
```

- Immutable objects (boolean, numbers, strings, tuples) cannot be changed once they are initialized.

```
>>> felix = 'Felix'
>>> felix = 'Felix'+' the cat.' # Create a new
>>>                             # string object.
```

# Testing for Equality

- The == operation tests for value equality.
- Works for all objects (objects of different type compare to `False`).

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a == b
True
```

```
>>> c = 1
>>> d = '1'
>>> c == d
False
```

- The '`is`' operator tests for object equality (i.e. if two variables are names for the same object).

```
>>> a is b
False
>>> a is None
False
```

# Comparators

- All comparator operations work for all objects:
  - Value equality: `==, !=, <, <=, >, >=`
  - Object equality: `is not`
- These operations return an object of type bool (either `True` or `False`)
- Can chain operations (expression is true if all pairs are true).

```
>>> a, b = 5, 7
>>> a >= 6
False
>>>  a < b <= 7 # Chained comparisons
True
```

# More Boolean Expressions

- Primitive literals `True` and `False` and result of comparator operations can be combined with boolean operations:
  - `not x`
  - `x and y`
  - `x or y`

```
>>> a = 3
>>> False or (a == 3)
True
>>> # () optional if priority is clear
>>> a > 0 and not False
True
```

# Some Operations on Numbers

- Binary: `x + y`, `x - y`, `x * y`, `x / y`,
  power `x ** y`, modulo `x % y`
- Unary: `-x`, `+x`
- Absolute value: `abs(x)`, Convert to integer: `int(x)`,
  Convert to float: `float(x)`

```
>>> x = -2 ** 4  - 4
>>> abs(x) # absolute value
20
>>> 20 / 3 # integer division
6
>>> 20 % 3 # modulo (remainder)
2
>>> float(20) / 3 # convert to float
6.666666666666667
```

- Can also convert (parse) strings.

```
>>> 5 + float('23.5')
>>> 28.5
```

# String/Integer Interning

```
>>> a = "hi"
>>> a is "h"+"i"
True
>>> b = "h"
>>> a is b+"i"
False
```

- This is a property of the cPython implementation:
  - Strings (and integers) are usually 'interned' (only one copy is kept in memory).
  - This speeds up string comparisons internally.
  - Can force interning explicitly:

```
>>> a is intern(b+"i")
True
```

- Once two strings have been created they will `always` be the same (strings are immutable).
- Thus it is unimportant if they are the same object or not.
- Strings (and integers) should never be compared using `is`. Use ==.

Course Description

About Python

Using Python

Data Types and Variables

## Control Flow

Container types

# Conditionals: `if` Statements

- If one `if` or `elif` matches the indented block statement is executed. Remaining conditions are ignored.
- `elif` and `else` are optional.
- If no `if` or `elif` matches, the indented block statement below `else` is executed.
- There is no `switch` statement in Python.

```
if conditionExp1:
    statement1
    ...
elif conditionExp2:
    statement2
    ...
elif confitionExp3:
    statement3
    ...
...
else:
    statement4
```

# Expressions in `if` and `elif` Conditions

Can use any expression as a condition ('in `boolean context`')

- int 0, `None`, empty string/list/tuple/dictionary/set → False
- any other object → True
- Boolean operations involving booleans and other objects.

```
print('type a number between 1 and 10!')
n = int(input()) # Read a number
if not n:
    print('Error: n was 0.')
elif n < 0:
    print('Error: n is negative.')
elif n == 1 or n == 2 or n == 3:
    print('n is prime.')
else:
    if n % 2 == 0 or n % 3 == 0:
        print('n is not prime.')
    else:
        print('n is prime.')
```

# Conditional Expressions / Ternary Operator

- `if` statements are often used to compute a value, depending on some condition.
- Need to store result in a variable and use additional lines.
- Concise way to write a conditional expression.

```
resultExp1 if conditionExp1 else resultExp2
```

```
>>> n = -1
>>> if n < 0:
...     result = n + 1
... else:
...     result = n - 1
>>> result
0
>>> # As conditional expression:
... n + 1 if n < 0 else n -1
0
```

# Loops: while Statements

Execute the indented statements repeatedly while conditionExp evaluates to True.

```
while conditionExp:
    statement
    ...
```

```
count = 0
while x > 0:
        x = x / 2
        count += 1
print('approximate log2:')
print(count)
```

# continue and break

'continue' interrupts the current iteration of the loop and continues at the next iteration.

```
>>> x = 5
>>> while x:
...     x -= 1
...     if not x % 2:
...         continue
...     print(x)
...
3
1
>>>
```

'break' interrupts the complete loop and continues execution below the loop.

```
>>> x = 10
>>> while True:
...     print(x)
...     x -= 1
...     if x == 7:
...         break
...
10
9
8
>>>
```

# Sequence Types

- Container objects that contain ordered sequences of elements:
  - String (a sequence of encoded characters) / Unicode.

```
x = 'Read me!  I'm a string!'
```

  - list (mutable sequence of objects)

```
x = [4, 8, 15, 16, 23, 42]
```

  - tuple (immutable sequence of objects)

```
x = (1.0, 'foo', (1,2,3))
```

- All sequence types support some common operations:
  - Get length. Concatenation and repetition.
  - Test for membership.
  - Access specific elements and 'slicing'.
  - Iterate through elements.

# Length of a Sequence / Concatenation and Repetition

- `len(x)` returns the length of sequence `x`.

```
>>> x = [] # Empty list
>>> len(x)
0
>>> Number of characters in string
... len('supercalifragilisticexpialidocious')
34
```

- `x + y` concatenates sequences `x` and `y`.
  - `x` and `y` need to have the same type.

```
>>> 'Hello' + 'World'
'HelloWorld'
```

- `x * n` or `n * x` repeats sequence `x` `n` times.

```
>>> 3 * ('A',)  # Single element tuple
('A', 'A', 'A')
```

# Testing for Sequence Membership

- `x in y` returns `True` if collection `y` contains object `x`, `False` otherwise.
  - Based on value equality (==).
  - `x not in y` is equivalent to `not x in y`

```
>>> 'coffee' in ['tea', 'coffee', 'juice']
True
```

- For strings only:
  - `in` also tests if `x` is a substring of `y`

```
>>> 'tuna' in fortunate
True
```

# Finding Index and Counting Elements

- `x.count(y)` returns the number of times y occurs in x.

```
>>> 'banana'.count('a')
3
>>> 'banana'.count('an') # also works for
    substrings
```

- `x.index(y)` returns the sequence index of the first occurrence of y.

```
>>> (23, 5, 8, 5).index(5)
1
```

# Sequence Indexing

- x[i] indexes the i'th element of sequence x (starting from 0).

```
>>> x = ((1, 2, 3), 'foo', 1.0)
>>> x[1]
'foo'
>>> x[0][2] # nested indexing
3
```

- reverse indexing starts at -1.

```
>>> x = ((1, 2, 3), 'foo', 1.0)
>>> x[-1]
1.0
```

# Sequence Slicing

- Slicing returns a copy of a subsequence.
- `x[i:j]` returns the subsequence from position i (inclusive) to position j (exclusive).
- `x[i:]` returns the subsequence from position i (inclusive) to the end.
- `x[:j]` returns the subsequence from the beginning to position j (exclusive).

```
>>> x = [0,1,2,3,4]
>>> x[1:]
[1,2,3,4]
>>> x[:-2] # can use reverse indexing
...        # in slice indices
[0,1,2]
>>> x[2:3]
[2]
```

# Iterating Through Sequences

Sequence data types implement the *iterator protocol*.

- Iterate through all elements of the sequence.
- Execute the statement with x bound to the current element.

```
for x in iterator:
    statement1
    ...
```

```
>>> for x in [1,2,3,4,5]:
...     if x % 2 == 0:
...         print(x)
...
2
4
```

- Can use `break` and `continue` in `for` loops.

# range

- `range(i)` produces an iterator of integers from 0 to `i` (exclusive).

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- `range(i,j)` produces an iterator over integers from `i` (inclusive) to `j` (exclusive).

```
>>> range(-3,4) # Can use negative indices
[-3, -2, -1, 0, 1, 2, 3]
```

- `range(i,j,s)` produces an iterator over integers from `i` (inclusive) to `j` (exclusive) in steps of `s`.

```
>>> range(10,1,-2) # Can use negative steps
[10, 8, 6, 4, 2]
```

# List Comprehension

- Perform some operation on each element of an iterator and get a new list.

```
[expr1 for x in sequence if condition]
```

```
>>> [x for x in range(10) if x % 2 == 0]
[0, 2, 4, 6, 8]
>>> [2 ** x for x in range(8)] # if is optional
1, 2, 4, 8, 16, 32, 64, 128
```

- Can use multiple for statements (e.g. compute all pairs)

```
>>> # Compute all pairs
... [(a,b) for a in range(1,3) for b in ['a','b']]
[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]
```

# `else` in List Comprehension

- Can use conditional expressions within a list comprehension.

```
>>> [a if a % 2 == 0 else 'bleep' \
...     for a in range(10)]
[0, 'bleep', 2, 'bleep', 4, 'bleep', 6, 'bleep',
    8, 'bleep']
```

- `if` does not filter the iteration in this case!

# List Operations (1)

- Lists are mutable and can be manipulated.

```
>>> a = ['apples', 'pears']
>>> a[1] = 'oranges'
>>> a
['apples', 'oranges']
```

- `list.append(x)` adds element x to the end of `list`
  - This is different from `list += [x]`, which creates a new list.

```
>>> a = ['apples', 'oranges']
>>> a.append('bananas')
>>> a
['apples', 'oranges', 'bananas']
```

- `list.pop()` removes the last element from `list` and returns it. Lists can be used as stacks.

```
>>> a.pop()
'bananas'
```

# List Operations (2)

- `list.remove(x)` removes the first occurrence of element `x` from the list.

```
>>> list = ['apple', 'orange', 'banana', 'orange']
>>> list.remove('orange')
['apple', 'banana', 'orange']
```

- `list.reverse()` reverses the order of the list.

```
>>> list.reverse()
>>> list
['orange', 'banana', 'apple']
```

- `list.sort()` sorts the list (using <=).

```
>>> list.sort() # alphabetically for strings.
>>> list
['apple', 'banana', 'orange']
```

# Dictionaries

- A dictionaries is a collections of objects indexed by unique keys.
- Most powerful built-in data structure.

```
>>> legs = {'cat':4, 'human':2, 'centipede':100}
>>> legs['cat']
4
```

- Assigning a new object to an unseen key inserts the key into the dictionary.

```
>>> legs['python'] = 0
```

- Keys are hashed (they need to be **immutable**).

# Testing for Membership

- `x in dict` returns `True` if x is a key of `dict`, `False` otherwise.

```
>>> legs = {'cat':4, 'human':2, 'python':0}
>>> 'cat' in legs
True
>>> 'centipede' not in legs
True
```

# Dictionary Items, Keys and Values (in Python 2.7)

- `dict.keys()` gets a list of keys.

```
>>> d = {'cat':4, 'human':2, 'elephant':4}
>>> d.keys()
['elephant', 'human', 'cat']
```

- `dict.values()` gets a list of dictionary values.

```
>>> d.values()
[4,2,4]
```

- `dict.items()` gets a list of `(key, value)` tuples.

```
>>> items = d.items()
>>> items
[('elephant', 4), ('human', 2), ('cat', 4)]
```

- `dict(x)` converts a list of items into a dictionary.

```
>>> dict(items)
{'cat': 4, 'human': 2, 'elephant': 4}
```

# Dictionary Items, Keys and Values (in Python 3)

- keys() and values() and items() gets a view of the dictionary.

```
>>> d = {'cat':4, 'human':2, 'elephant':4}
>>> keys = d.keys()
>>> keys
dict_keys(['elephant', 'human', 'cat'])
```

- If the dictionary changes, the view changes too.

```
>>> d['penguin'] = 2
>>> keys
dict_keys(['penguin', 'elephant', 'human', '
    cat'])
```

# Sets

Sets (mutable) / frozensets (immutable) are unordered bags of unique objects.

```
>>> s = set(['apple', 'banana', 'orange'])
```

- Set membership: `x in s`

```
>>> 'apple' in s
True
>>> 'apple' not in s
False
```

- is `s` a subset/superset of `t`?

```
>>> frozenset(['apple','orange']) <= s
True
>>> # alternative syntax (2.7 and 3.x)
... {'apple','orange','banana','mango'} >= s
True
```

# Sets - Union/Intersection/Difference

- get union of s and t as a new set.

```
>>> set(['a', 'b']) | set(['c','d'])
set(['a', 'b', 'c', 'd'])
```

- get intersection of s and t as a new set.

```
>>> set(['a', 'b']) & set(['a','c'])
set(['a'])
```

- get difference between s and t as a new set.

```
>>> set(['a', 'b']) - set(['a','c'])
set(['b'])
```

# Mutable Sets - update, add, remove

These operations do not work for frozensets:

- add all elements of set  to set s

```
>>> s = set(['a', 'b'])
>>> s.update(set(['a','c']))
>>> s
set(['a', 'b', 'c'])
```

- add object x to s.

```
>>> s.add('d')
>>> s
set(['a', 'b', 'c','d'])
```

- remove object x from s.

```
>>> s.remove('b')
>>> s
set(['a','c','d'])
```

# Homework 1

- Available on course website. Due: Tue Sep 16, 11:59pm.
- Submit on Courseworks (not by email).
- Submission instructions and guidelines on course website.