Why Python?

Richard Killam

Sunday 27th September, 2015

Outline

Python Success Stories

- Civ IV
- Eve
- Dropbox
- Google
- IceCube
- LucasFilm
- NASA
- Reddit
- Ubuntu

Biggest Motivator

Biggest Motivator

It’s Free!

&

Open Source!

PEP 8 Rules

- snake_case_everything
- except CapitalCamelCase class names
- _private_variable
- __hidden_private_variable (Can't be seen by getattr)
- avoid_keyword_conflict_
- __magic_functions__ (Not as magical as advertised)
- Indentation, use spaces in multiples of 4 (4, 8, 12, ...)
- And many others (it's a really long list)

PEP 8 Results

PHP v.s. Python

- strstr(string, substring)

PEP 8 Results

PHP v.s. Python

- strstr(string, substring) v.s. string.index(substring)

PEP 8 Results

PHP v.s. Python

- strstr(string, substring) v.s. string.index(substring)
- str_replace(substring, replace_with, string)

PEP 8 Results

PHP v.s. Python

- strstr(string, substring) v.s. string.index(substring)
- str_replace(substring, replace_with, string) v.s. string.replace(substring, replace_with)

PEP 8 Results

PHP v.s. Python

- strstr(string, substring) v.s. string.index(substring)
- str_replace(substring, replace_with, string) v.s. string.replace(substring, replace_with)
- htmlspecialchars_decode
- get_html_translation_table

Batteries

- Over 500 built-in libraries
- High performance wrappers of C libraries
- PIP: **P**ip **I**nstalls **P**ython has over 100 easy to install packages
- Github has 102,178 repositories written in Python, for Python

  Github user vinta compiled a list of awesome python packages:
  `https://github.com/vinta/awesome-python`

Whitespace

What does this do?

C-style (Java with printf):

```
int s = 0;
for(int i = 0; i < 10; ++i)
    printf("%d + %d\n", s, i);
  s += i;
```

Python:

Whitespace

What does this do?

C-style (Java with printf):

```
int s = 0;
for(int i = 0; i < 10; ++i)
   printf("%d + %d\n", s, i);
  s += i;
```

Python:

```
s = 0;
for i in xrange(10):
   print("%d + %d\n" % (s, i))
  s += i
```

self

What does this do?

C-style (Java with printf):

```
class Test {
    private int x;
    public Test(int x) {
        this.x = 2;
        printf("%d\n", x);
    }
    public void printX() {
        printf("%d\n", x);
    }
}
```

Python:

self

What does this do?

C-style (Java with printf):

```
class Test {
    private int x;
    public Test(int x) {
        this.x = 2;
        printf("%d\n", x);
    }
    public void printX() {
        printf("%d\n", x);
    }
}
```

Python:

```
class Test(object):

    def __init__(self, x):
        self.x = 2
        print(x)

    def print_x(self):
        print(self.x)
```

Looks like **Math**, Reads like **English**

- (In)equality Chaining:
  - $1 < x < y < z < 10$
  - $x = y = z = 2$

Looks like **Math**, Reads like **English**

- (In)equality Chaining:
  - $1 < x < y < z < 10$
  - $x = y = z = 2$
- `for(int i = 0; i < my_list.length; ++i)`

  v.s.

  `for variable in my_list`

Looks like **Math**, Reads like **English**

- (In)equality Chaining:
  - $1 < x < y < z < 10$
  - $x = y = z = 2$
- `for(int i = 0; i < my_list.length; ++i)`

  v.s.

  `for variable in my_list`
- `if x in my_list`

  Still O(n) but incredibly well optimized

List Creation

- List multiplication (works for strings too)

```
my_list = [[1] * 2] * 2 ⇒ [[1, 1],
                            [1, 1]]
```

List Creation

- List multiplication (works for strings too)

```
my_list = [[1] * 2] * 2 ⇒ [[1, 1],
                           [1, 1]]
```

- List comprehension

```
[2**i for i in xrange(5)] ⇒ [1, 2, 4, 8, 16]
```

List comprehension v.s. filter, lambda, map, reduce

-
  ```
  python -mtimeit -s'l=range(10)' 'map(lambda x:x+2,l)'
      100000 loops, best of 3: 4.24 usec per loop
  ```

  ```
  python -mtimeit -s'l=range(10)' '[x+2 for x in l]'
      100000 loops, best of 3: 2.32 usec per loop
  ```

List comprehension v.s. filter, lambda, map, reduce

-
  ```
  python -mtimeit -s'l=range(10)' 'map(lambda x:x+2,l)'
      100000 loops, best of 3: 4.24 usec per loop

  python -mtimeit -s'l=range(10)' '[x+2 for x in l]'
      100000 loops, best of 3: 2.32 usec per loop
  ```
- Comprehensions more math like: set notation $= \{x + 2 \mid \forall x \epsilon l\}$
- More consistent with english:

  "Map item + 2 for all items in l"

  v.s.

  "I want a list of all of the items in l plus 2"

List Indexing

- my_list[my_list.length - 1]

List Indexing

- my_list[my_list.length - 1] v.s. my_list[-1]

List Indexing

- my_list[my_list.length - 1] v.s. my_list[-1]
- `for(int i = 1; i < my_list.length; i += 2)`

List Indexing

- my_list[my_list.length - 1] v.s. my_list[-1]
- `for(int i = 1; i < my_list.length; i += 2)`

  v.s. my_list[1::2]

List Indexing

- my_list[my_list.length - 1] v.s. my_list[-1]
- `for(int i = 1; i < my_list.length; i += 2)`

  v.s. my_list[1::2]
- range(5)[1:] ⇒ [1, 2, 3, 4]
  range(5)[:-1] ⇒ [0, 1, 2, 3]
  range(5)[1:3] ⇒ [1, 2]
  range(10)[::2] ⇒ [0, 2, 4, 6, 8]
  range(10)[1::2] ⇒ [1, 3, 5, 7, 9]

Sets

- Remove duplicates:

  `set([1, 2, 3, 4, 1, 2, 3, 4])` $\Rightarrow$ `{1, 2, 3, 4}`

Sets

- Remove duplicates:

  `set([1, 2, 3, 4, 1, 2, 3, 4])` ⇒ `{1, 2, 3, 4}`

- `if x in my_set:`

Avg:	**O(1)**
Worst:	O(n)

Set Operations

- set1 - set2 == set1.difference(set2)
  O(len(set1))

Set Operations

- set1 - set2 == set1.difference(set2)
  O(len(set1))
- set1 & set2 == set1.intersection(set2)
  Avg: O(min(len(set1, set2)))

Set Operations

- set1 - set2 == set1.difference(set2)
  O(len(set1))
- set1 & set2 == set1.intersection(set2)
  Avg: O(min(len(set1, set2)))
- set1 | set2 == set1.union(set2)
  O(len(set1)+len(set2))

Set Operations

- set1 - set2 == set1.difference(set2)
  O(len(set1))
- set1 & set2 == set1.intersection(set2)
  Avg: O(min(len(set1, set2)))
- set1 | set2 == set1.union(set2)
  O(len(set1)+len(set2))
- set1 ^ set2 == set1.symmetric_difference(set2)
  Avg: O(len(set1))

Dictionaries

- Built-in hash map
- O(1) lookups:

```
    my_dict.get(key) # None if key does not exist
```

- dict comprehension:

```
{obj.name: obj.data for obj in my_objs}
```

- Easy looping:

```
for key, val in my_dict.items():
    print('{}: {}'.format(key, val))
```

What are Context Managers?

Handles allocation and release of a resource

This:

```
f = None
try:
    f = open('f.txt', 'r')
    # Do stuff...
finally:
    if f is not None:
        f.close()
```

Becomes:

What are Context Managers?

Handles allocation and release of a resource

This:

```
f = None
try:
    f = open('f.txt', 'r')
    # Do stuff...
finally:
    if f is not None:
        f.close()
```

Becomes:

```
with open('f.txt', 'r')
    as f:
    # Do stuff...
```

Why use Context Managers?

- Avoid verbose repeat code
- Ensure release is handled properly
- Variable scope retention

Context Manager Uses

- Ensure successful db transaction before commit
- Holding some I/O
- Locking a thread
- Opening a file

What are Decorators?

Classes or higher order functions that wrap a given function or class

This:

```
def my_f(...):
    Do stuff...
    ret = f(...)
    Do more stuff...
    return ret
```

Becomes:

What are Decorators?

Classes or higher order functions that wrap a given function or class

This:

```
def my_f(...):
    Do stuff...
    ret = f(...)
    Do more stuff...
    return ret
```

Becomes:

```
@my_decorator
def f(...):
    ...
```

Why use Decorators?

- Avoid verbose repeat code
- Closures allow for state retention:

  aggregation, memoization, etc

Decorator Uses

- Argument/return checking
- Function timeout
- Logging (decorate class)
- Memoization
- Thunkifying (Parallelizing)

What are Generators?

An easy way to support iterations

This:

```
class Test(object):
    def __init__(sf,s,e):
        sf.c = s
        sf.e = e
    def __iter__(sf):
        return sf
    def next(sf):
        if sf.c>=sf.e:
            raise StopIter
        r = sf.c
        sf.c += 1
        return r
```

Becomes:

What are Generators?

An easy way to support iterations

This:

```
class Test(object):
    def __init__(sf,s,e):
        sf.c = s
        sf.e = e
    def __iter__(sf):
        return sf
    def next(sf):
        if sf.c>=sf.e:
            raise StopIter
        r = sf.c
        sf.c += 1
        return r
```

Becomes:

```
def test(start, end):
    while start < end:
        yield start
        start += 1
```

Why use Generators?

- Lazy evaluation
- Less memory usage

Generator Uses

- Co-routines (producer/consumer) using two-way generators
- Interpolations/regressions (unknown number of iterations)
- Process text files

IPython

Featureful Python REPL

IPython

Featureful Python REPL

- Explore objects (my_obj?)
- Magic functions (debug, edit, run, timeit, ...)
- Multi shell support (bash, javascript, latex, perl, pypy, ruby)
- Notebooks allow for fast and easy sharing of code and data

Scientific Computing Libraries

- **numpy**: Flexible array structures for fast mathematical operations
- **pandas**: A powerful data analysis and manipulation library
- **scipy**: 32 subpackages for scientific and mathematical operations
- **scikit-learn (sklearn)**: Easy to use machine learning library
- **nltk**: Massive natural language processing library
- **cv/cv2**: Python wrappers for the fast and powerful OpenCV computer vision library
- **matplotlib**: Easy to use plotting library
- **pickle**: Object serializing

Other Features (Python Buzzwords)

- **Virtualenv**:
  Encapsulate python projects and their dependencies
- **MicroPython**: Python for micro controllers
- **Namespaces**: Built-in way to encapsulate your modules
- **Cython and PyPy**: Python speed boosts
- **CPython, Jython, IronPython, RPython, pyjs**:
  Multi-language support (C, Java, .NET, R, JavaScript, ...)
- **Brython**: Python in the browser
- **Django, Flask, Bottle**: Python on the server

The Zen of Python, by Tim Peters: import this

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one– and preferably only one –obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!