The Pythonic Way

Jack Rosenthal

December 5, 2016

Mines Linux Users Group

Table of contents

Background

A Bit of History

- Python first appeared in early 1991. *This means that Python is older than Java and Ruby.*
- Guido van Rossum (GvR, the creator of Python) designed his language with **emphasis on readability**.
- Python was named after *Monty Python's Flying Circus*.
- The language quickly gained popularity because of its appeal to long-time UNIX/C hackers[1].

[1]See the Jargon File on hackers vs. crackers

A Bit of History

- Python first appeared in early 1991. *This means that Python is older than Java and Ruby.*
- Guido van Rossum (GvR, the creator of Python) designed his language with **emphasis on readability**.
- Python was named after *Monty Python's Flying Circus*.
- The language quickly gained popularity because of its appeal to long-time UNIX/C hackers[1].

[1] See the Jargon File on hackers vs. crackers

A Bit of History

- Python first appeared in early 1991. *This means that Python is older than Java and Ruby.*
- Guido van Rossum (GvR, the creator of Python) designed his language with **emphasis on readability**.
- Python was named after *Monty Python's Flying Circus*.
- The language quickly gained popularity because of its appeal to long-time UNIX/C hackers[1].

[1]See the Jargon File on hackers vs. crackers

A Bit of History

- Python first appeared in early 1991. *This means that Python is older than Java and Ruby.*
- Guido van Rossum (GvR, the creator of Python) designed his language with **emphasis on readability**.
- Python was named after *Monty Python's Flying Circus*.
- The language quickly gained popularity because of its appeal to long-time UNIX/C hackers[1].

[1]See the Jargon File on hackers vs. crackers

Python: The Perfect Companion to Linux

- Easy to use without a domain specific IDE or editor
- On-line documentation with `pydoc` similar to `man`
- UNIX system calls share the same name as in `libc`
- Fast and lightweight
- Very general purpose
- Easy to learn, but plenty to master

Python: The Perfect Companion to Linux

- Easy to use without a domain specific IDE or editor
- On-line documentation with `pydoc` similar to `man`
- UNIX system calls share the same name as in `libc`
- Fast and lightweight
- Very general purpose
- Easy to learn, but plenty to master

Python: The Perfect Companion to Linux

- Easy to use without a domain specific IDE or editor
- On-line documentation with `pydoc` similar to `man`
- UNIX system calls share the same name as in `libc`
- Fast and lightweight
- Very general purpose
- Easy to learn, but plenty to master

Python: The Perfect Companion to Linux

- Easy to use without a domain specific IDE or editor
- On-line documentation with `pydoc` similar to `man`
- UNIX system calls share the same name as in `libc`
- Fast and lightweight
- Very general purpose
- Easy to learn, but plenty to master

Python: The Perfect Companion to Linux

- Easy to use without a domain specific IDE or editor
- On-line documentation with `pydoc` similar to `man`
- UNIX system calls share the same name as in `libc`
- Fast and lightweight
- Very general purpose
- Easy to learn, but plenty to master

Python: The Perfect Companion to Linux

- Easy to use without a domain specific IDE or editor
- On-line documentation with `pydoc` similar to `man`
- UNIX system calls share the same name as in `libc`
- Fast and lightweight
- Very general purpose
- Easy to learn, but plenty to master

A Note on Python 2 and Python 3

Python 3 fixed many odds and ends from older versions of Python. When it was originally released, its usage was low due to many backwards incompatibilities. Now days, most modern projects use Python 3, so this issue is largely irrelevant.

For the purposes of this presentation, we will be talking *strictly of Python 3*.

Setting The Default

Some systems have Python 2 as the default, the general solution is to alias `python` to `python3`. Add this to your shell's *rc file*.

```
alias python=python3
```

A Note on Python 2 and Python 3

Python 3 fixed many odds and ends from older versions of Python. When it was originally released, its usage was low due to many backwards incompatibilities. Now days, most modern projects use Python 3, so this issue is largely irrelevant.

For the purposes of this presentation, we will be talking *strictly of Python 3*.

Setting The Default

Some systems have Python 2 as the default, the general solution is to alias `python` to `python3`. Add this to your shell's *rc file*.

```
alias python=python3
```

A Simple Example

The classical *Fizz Buzz* problem: for all the numbers from 1 to 100, print `Fizz` if the number is divisible by 3, `Buzz` if divisible by 5, `Fizz Buzz` if divisible by 3 and 5, and the number otherwise.

```
for i in range(1, 101):
    if i % 3 == 0 and i % 5 == 0:
        print("Fizz Buzz")
    elif i % 3 == 0:
        print("Fizz")
    elif i % 5 == 0:
        print("Buzz")
    else:
        print(i)
```

The Zen of Python

PEP-20 lists a series of principles for which the language was designed under. Typing `import this` at the Python interpreter will show you the zen:

```
>>> import this
The Zen of Python, by Tim Peters
...
```

In short, Python was designed with careful thought about how to make it *Pythonic*.

The Zen of Python

PEP-20 lists a series of principles for which the language was designed under. Typing `import this` at the Python interpreter will show you the zen:

```
>>> import this
The Zen of Python, by Tim Peters
...
```

In short, Python was designed with careful thought about how to make it *Pythonic*.

Python Style

A Foolish Consistency is the Hobgoblin of Little Minds

- GvR makes a point: *code is read more often than it is written*, so **readability counts**.
- Python is one of the few languages with a style guide (PEP-8) since there is a huge amount of Python code out there and the language's core principle is readability.
- Thus, it's important to follow Python's official style whenever possible

Legacy Code

It should be noted that when working on a project that was started before the ages of PEP-8, generally they have their own style guide and you should follow that instead. Otherwise, it would be generally considered unacceptable to not follow PEP-8.

A Foolish Consistency is the Hobgoblin of Little Minds

- GvR makes a point: *code is read more often than it is written*, so **readability counts**.
- Python is one of the few languages with a style guide (PEP-8) since there is a huge amount of Python code out there and the language's core principle is readability.
- Thus, it's important to follow Python's official style whenever possible

Legacy Code

It should be noted that when working on a project that was started before the ages of PEP-8, generally they have their own style guide and you should follow that instead. Otherwise, it would be generally considered unacceptable to not follow PEP-8.

A Foolish Consistency is the Hobgoblin of Little Minds

- GvR makes a point: *code is read more often than it is written*, so **readability counts**.
- Python is one of the few languages with a style guide (PEP-8) since there is a huge amount of Python code out there and the language's core principle is readability.
- Thus, it's important to follow Python's official style whenever possible

Legacy Code

It should be noted that when working on a project that was started before the ages of PEP-8, generally they have their own style guide and you should follow that instead. Otherwise, it would be generally considered unacceptable to not follow PEP-8.

A Foolish Consistency is the Hobgoblin of Little Minds

- GvR makes a point: *code is read more often than it is written*, so **readability counts**.
- Python is one of the few languages with a style guide (PEP-8) since there is a huge amount of Python code out there and the language's core principle is readability.
- Thus, it's important to follow Python's official style whenever possible

Legacy Code

It should be noted that when working on a project that was started before the ages of PEP-8, generally they have their own style guide and you should follow that instead. Otherwise, it would be generally considered unacceptable to not follow PEP-8.

Naming

- Python uses `snake_case` for variable names, function names, method names, and module names
- You should avoid using underscores when possible to improve readability (eg. `randint` is better than `rand_int`, as the naming is obvious without the underscore).
- When there are conflicts with builtin keywords and a better name is not possible, an underscore should be appended to the variable name (eg. `class_`)
- Class names should be typed in `CapWords`
- Function, method, and class names should describe the interface rather than the implementation.
- Private methods and variables should start with an underscore.

Naming

- Python uses `snake_case` for variable names, function names, method names, and module names
- You should avoid using underscores when possible to improve readability (eg. `randint` is better than `rand_int`, as the naming is obvious without the underscore).
- When there are conflicts with builtin keywords and a better name is not possible, an underscore should be appended to the variable name (eg. `class_`)
- Class names should be typed in `CapWords`
- Function, method, and class names should describe the interface rather than the implementation.
- Private methods and variables should start with an underscore.

Naming

- Python uses `snake_case` for variable names, function names, method names, and module names
- You should avoid using underscores when possible to improve readability (eg. `randint` is better than `rand_int`, as the naming is obvious without the underscore).
- When there are conflicts with builtin keywords and a better name is not possible, an underscore should be appended to the variable name (eg. `class_`)
- Class names should be typed in `CapWords`
- Function, method, and class names should describe the interface rather than the implementation.
- Private methods and variables should start with an underscore.

Naming

- Python uses `snake_case` for variable names, function names, method names, and module names
- You should avoid using underscores when possible to improve readability (eg. `randint` is better than `rand_int`, as the naming is obvious without the underscore).
- When there are conflicts with builtin keywords and a better name is not possible, an underscore should be appended to the variable name (eg. `class_`)
- Class names should be typed in `CapWords`
- Function, method, and class names should describe the interface rather than the implementation.
- Private methods and variables should start with an underscore.

Naming

- Python uses `snake_case` for variable names, function names, method names, and module names
- You should avoid using underscores when possible to improve readability (eg. `randint` is better than `rand_int`, as the naming is obvious without the underscore).
- When there are conflicts with builtin keywords and a better name is not possible, an underscore should be appended to the variable name (eg. `class_`)
- Class names should be typed in `CapWords`
- Function, method, and class names should describe the interface rather than the implementation.
- Private methods and variables should start with an underscore.

Naming

- Python uses `snake_case` for variable names, function names, method names, and module names
- You should avoid using underscores when possible to improve readability (eg. `randint` is better than `rand_int`, as the naming is obvious without the underscore).
- When there are conflicts with builtin keywords and a better name is not possible, an underscore should be appended to the variable name (eg. `class_`)
- Class names should be typed in `CapWords`
- Function, method, and class names should describe the interface rather than the implementation.
- Private methods and variables should start with an underscore.

Indentation

As Python uses the indentation of the text to denote scope, consistency of indentation is critically important. PEP-8 recommends the following:

- Use 4 spaces per indentation level, **never use hard tabs**.
- On multiline function calls, list literals, etc., the arguments should be aligned and indented from the rest of the text. “Hanging indent” is acceptable as well.
- Multiline `if`/`while` etc. should be indented to align with the top line

Indentation

As Python uses the indentation of the text to denote scope, consistency of indentation is critically important. PEP-8 recommends the following:

- Use 4 spaces per indentation level, **never use hard tabs**.
- On multiline function calls, list literals, etc., the arguments should be aligned and indented from the rest of the text. "Hanging indent" is acceptable as well.
- Multiline `if`/`while` etc. should be indented to align with the top line

Indentation

As Python uses the indentation of the text to denote scope, consistency of indentation is critically important. PEP-8 recommends the following:

- Use 4 spaces per indentation level, **never use hard tabs**.
- On multiline function calls, list literals, etc., the arguments should be aligned and indented from the rest of the text. "Hanging indent" is acceptable as well.
- Multiline `if`/`while` etc. should be indented to align with the top line

Other Pet Peeves

- Keep lines to 79 characters[2]
- Avoid extraneous whitespace inside parentheses, brackets, and braces

```
Yes: spam(ham[1], {eggs: 2})
No:  spam( ham[ 1 ], { eggs: 2 } )
```

- Don't use parentheses on `if`/`while` etc. like you might in C-like languages

```
Yes: if i < 3:
No:  if(i < 3):
```

[2]It's OK to go to 90 or 100 if everyone in your project agrees.

Other Pet Peeves

- Keep lines to 79 characters[2]
- Avoid extraneous whitespace inside parentheses, brackets, and braces

```
Yes: spam(ham[1], {eggs: 2})
No:  spam( ham[ 1 ], { eggs: 2 } )
```

- Don't use parentheses on `if`/`while` etc. like you might in C-like languages

```
Yes: if i < 3:
No:  if(i < 3):
```

[2] It's OK to go to 90 or 100 if everyone in your project agrees.

Other Pet Peeves

- Keep lines to 79 characters[2]
- Avoid extraneous whitespace inside parentheses, brackets, and braces

```
Yes: spam(ham[1], {eggs: 2})
No:  spam( ham[ 1 ], { eggs: 2 } )
```

- Don't use parentheses on `if`/`while` etc. like you might in C-like languages

```
Yes: if i < 3:
No:  if(i < 3):
```

[2] It's OK to go to 90 or 100 if everyone in your project agrees.

Truthiness

Anything `None`, `False`, zero, or an empty sequence/mapping will implicity be false, and you *should* take advantage of that.

```
Disgusting: if mybool == False:
Pythonic:   if mybool:

Disgusting: if mydata == None:
Pythonic:   if mydata:

Ehh:        if mynumber != 0:
Pythonic:   if mynumber:

Ugly:       if len(mylist) == 0:
Better:     if not len(mylist):
Pythonic:   if not mylist:
```

Comments

Every comment in the source code is a personal failure of the programmer, because it proves that he didn't manage to express the purpose of the code fragment with the programming language itself. — Uncle Bob

Take Home: Comments are important when they are needed, but you should try and make your code readable instead.

Readability Counts!

No really, it is of utmost importance that Python code be readable **by following the guidelines of PEP-8**. You should read through PEP-8 before getting serious with Python.

Language Structures

Literals

```python
# List
names = ['Euclid', 'Lovelace', 'Turing']

# Tuples (immutible)
names = ('Euclid', 'Lovelace', 'Turing')

# To specify a one-tuple
names = ('Euclid',)

# Dictionaries
names = {'Lovelace': 'Ada', 'Turing': 'Alan'}

# Sets (unique values)
names = {'Euclid', 'Lovelace', 'Turing'}
```

Selection

Python's primary structure for selection is `if`:

```
if i == 0 and j == 1:
    print(i, j)
elif i > 10 or j < 0:
    print("whoa!")
else:
    print("all is fine")
```

There's also a ternary operator (good for simple conditionals):

```
def foo(bar, baz):
    return bar if bar else baz
```

Selection

Python's primary structure for selection is `if`:

```
if i == 0 and j == 1:
    print(i, j)
elif i > 10 or j < 0:
    print("whoa!")
else:
    print("all is fine")
```

There's also a ternary operator (good for simple conditionals):

```
def foo(bar, baz):
    return bar if bar else baz
```

Why no `switch/case`?

Most `switch`/`case` statements over-complicate what could be done in a single line using a dictionary. Where this is not the case, you really shouldn't be using a `switch` anyway.

Why no switch/case?

An Example switch in C

```c
switch (a) {
    case 'q':
        count++;
        break;
    case 'x':
        count--;
        break;
    case 'z':
        count += 4;
}
```

Why no `switch/case`?

The Pythonic Way

```
choice = {'q': 1, 'x': -1, 'z': 4}
count += choice[a]
```

Iteration

Python provides your traditional `while` loop, the syntax is similar to `if`:

```
while n < 100:
    j /= n
    n += j
```

But under most cases, the **range-based `for`** loop is preferred:

```
for x in mylist:
    print(x)
```

It should be noted that Python's `for` loop is strictly range-based, unlike C's `for` loop which is really just a fancy `while` loop.

Iteration

Python provides your traditional `while` loop, the syntax is similar to `if`:

```
while n < 100:
    j /= n
    n += j
```

But under most cases, the **range-based `for`** loop is preferred:

```
for x in mylist:
    print(x)
```

It should be noted that Python's `for` loop is strictly range-based, unlike C's `for` loop which is really just a fancy `while` loop.

`while-else` and `for-else`

Little known is the ability to pair an `else` block with `for` and `while`. The block will be executed *only if* the loop finishes **without breaking**.

An example of this can be seen below:

```
for i in range(10):
    x = input("Enter your guess: ")
    if i == x:
        print("You win!")
        break
else:
    print("Truly incompetent!")
```

Slicing

```
mylist = [1, 2, 3, 4]

# syntax is [start:stop:step], step optional
mylist[1:3] # => [2, 3]

# unused parameters can be ommited
mylist[::-1] # => [4, 3, 2, 1]

# without the first element
mylist[1:] # => [2, 3, 4]

# without the last element
mylist[:-1] # => [1, 2, 3]
```

Tuple Expansion & Collection

Multiple assignments work like so:

```
names = ("R. Stallman", "L. Torvalds", "B. Joy")
a, b, c = names
```

* can be used to collect a tuple:

```
# drop the lowest and highest grade
grades = (79, 81, 93, 95, 99)
lowest, *grades, highest = grades
```

The same can be done to expand a tuple in a function call:

```
print(*grades)
```

Tuple Expansion & Collection

Multiple assignments work like so:

```
names = ("R. Stallman", "L. Torvalds", "B. Joy")
a, b, c = names
```

`*` can be used to collect a tuple:

```
# drop the lowest and highest grade
grades = (79, 81, 93, 95, 99)
lowest, *grades, highest = grades
```

The same can be done to expand a tuple in a function call:

```
print(*grades)
```

Tuple Expansion & Collection

Multiple assignments work like so:

```
names = ("R. Stallman", "L. Torvalds", "B. Joy")
a, b, c = names
```

`*` can be used to collect a tuple:

```
# drop the lowest and highest grade
grades = (79, 81, 93, 95, 99)
lowest, *grades, highest = grades
```

The same can be done to expand a tuple in a function call:

```
print(*grades)
```

Functions

Functions are *first-class citizens* in Python:

```
>>> def identity(x):
...     return x
...
>>> type(identity)
<class 'function'>
```

Functions can also be written anonymously as `lambda`s:

```
>>> identity = lambda x:x
>>> identity(42)
42
```

In this case, the first style is preferred. It's a bit easier to read, not to mention it's actually named.

Functions

Functions are *first-class citizens* in Python:

```
>>> def identity(x):
...     return x
...
>>> type(identity)
<class 'function'>
```

Functions can also be written anonymously as `lambda`s:

```
>>> identity = lambda x:x
>>> identity(42)
42
```

In this case, the first style is preferred. It's a bit easier to read, not to mention it's actually named.

Functions

Functions are *first-class citizens* in Python:

```
>>> def identity(x):
...     return x
...
>>> type(identity)
<class 'function'>
```

Functions can also be written anonymously as `lambda`s:

```
>>> identity = lambda x:x
>>> identity(42)
42
```

In this case, the first style is preferred. It's a bit easier to read, not to mention it's actually named.

*args, **kwargs

Python allows you to define functions that take a variable number of positional (`*args`) or keyword (`**kwargs`) arguments. In principle, this really just works like tuple expansion/collection.

```python
def crazyprinter(*args, **kwargs):
    for arg in args:
        print(arg)
    for k, v in kwargs.items():
        print("{}={}".format(k, v))

crazyprinter("hello", "cheese", bar="foo")
# hello
# cheese
# bar=foo
```

*args, **kwargs

Python allows you to define functions that take a variable number of positional (`*args`) or keyword (`**kwargs`) arguments. In principle, this really just works like tuple expansion/collection.

```
def crazyprinter(*args, **kwargs):
    for arg in args:
        print(arg)
    for k, v in kwargs.items():
        print("{}={}".format(k, v))

crazyprinter("hello", "cheese", bar="foo")
# hello
# cheese
# bar=foo
```

Generator Functions

Python provides a special kind of function which `yield`s rather than `return`s. This **generator function** is effectively an efficient iterable.

Consider the `range` function we have been using[3]:

```
def range(start, stop, step=1):
    i = 0
    while i < stop:
        yield i
        i += step
```

As we will see later on, generator functions are a certain kind of the more generic **generator**.

[3] This is actually a simplification

Generator Functions

Python provides a special kind of function which `yield`s rather than `return`s. This **generator function** is effectively an efficient iterable.

Consider the `range` function we have been using[3]:

```
def range(start, stop, step=1):
    i = 0
    while i < stop:
        yield i
        i += step
```

As we will see later on, generator functions are a certain kind of the more generic **generator**.

[3]This is actually a simplification

Object Oriented Programming

Classes

A simple class can be defined like so:

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
```

A few things to notice:

- `__init__` is the initializes the object. It's actually what is called a **magic method**
- All the methods of the class take a parameter `self`, the object you are working on

Classes

A simple class can be defined like so:

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
```

A few things to notice:

- `__init__` is the initializes the object. It's actually what is called a **magic method**
- All the methods of the class take a parameter `self`, the object you are working on

Magic Methods

Magic methods are methods with certain names that allow you to bind features of your class to certain Python features.

- `__init__` was the simple example we just saw.
- `__del__` gets called when your object gets destructed.
- `__lt__`, `__eq__`, etc. allow you to define comparisons.
- `__len__` binds into Python's `len(·)`
- There's far more than I can mention here. Read the docs!

Why do this rather than `.equals()`, `.length()` and such?

In the face of ambiguity, refuse the temptation to guess. There should be one – and preferably only one – obvious way to do it.

Avoid `.length()`, `.getLength()`, `.size()` inconsistencies

Magic Methods

Magic methods are methods with certain names that allow you to bind features of your class to certain Python features.

- `__init__` was the simple example we just saw.
- `__del__` gets called when your object gets destructed.
- `__lt__`, `__eq__`, etc. allow you to define comparisons.
- `__len__` binds into Python's `len(·)`
- There's far more than I can mention here. Read the docs!

Why do this rather than `.equals()`, `.length()` and such?

In the face of ambiguity, refuse the temptation to guess. There should be one – and preferably only one – obvious way to do it.

Avoid `.length()`, `.getLength()`, `.size()` inconsistencies

Magic Methods

Magic methods are methods with certain names that allow you to bind features of your class to certain Python features.

- `__init__` was the simple example we just saw.
- `__del__` gets called when your object gets destructed.
- `__lt__`, `__eq__`, etc. allow you to define comparisons.
- `__len__` binds into Python's `len(·)`
- There's far more than I can mention here. Read the docs!

Why do this rather than `.equals()`, `.length()` and such?

In the face of ambiguity, refuse the temptation to guess. There should be one – and preferably only one – obvious way to do it.

Avoid `.length()`, `.getLength()`, `.size()` inconsistencies

Magic Methods

Magic methods are methods with certain names that allow you to bind features of your class to certain Python features.

- `__init__` was the simple example we just saw.
- `__del__` gets called when your object gets destructed.
- `__lt__`, `__eq__`, etc. allow you to define comparisons.
- `__len__` binds into Python's `len(·)`
- There's far more than I can mention here. Read the docs!

Why do this rather than `.equals()`, `.length()` and such?

In the face of ambiguity, refuse the temptation to guess. There should be one – and preferably only one – obvious way to do it.

Avoid `.length()`, `.getLength()`, `.size()` inconsistencies

Magic Methods

Magic methods are methods with certain names that allow you to bind features of your class to certain Python features.

- `__init__` was the simple example we just saw.
- `__del__` gets called when your object gets destructed.
- `__lt__`, `__eq__`, etc. allow you to define comparisons.
- `__len__` binds into Python's `len(·)`
- There's far more than I can mention here. Read the docs!

Why do this rather than `.equals()`, `.length()` and such?

In the face of ambiguity, refuse the temptation to guess. There should be one – and preferably only one – obvious way to do it.

Avoid `.length()`, `.getLength()`, `.size()` inconsistencies

Magic Methods

Magic methods are methods with certain names that allow you to bind features of your class to certain Python features.

- `__init__` was the simple example we just saw.
- `__del__` gets called when your object gets destructed.
- `__lt__`, `__eq__`, etc. allow you to define comparisons.
- `__len__` binds into Python's `len(·)`
- There's far more than I can mention here. Read the docs!

Why do this rather than `.equals()`, `.length()` and such?

In the face of ambiguity, refuse the temptation to guess. There should be one – and preferably only one – obvious way to do it.

Avoid `.length()`, `.getLength()`, `.size()` inconsistencies

Properties

Readability counts, so Python provides a way to avoid writing "getters and setters" when unnecessary.

In Java, it's nearly impossible to make everything public, since changing a class to use getters and setters would require a change of everything that interfaces with it.

Python's properties allow you to make your variable public to begin with, and then write getters and setters only once they are needed to actually check something.

Properties

Readability counts, so Python provides a way to avoid writing "getters and setters" when unnecessary.

In Java, it's nearly impossible to make everything public, since changing a class to use getters and setters would require a change of everything that interfaces with it.

Python's properties allow you to make your variable public to begin with, and then write getters and setters only once they are needed to actually check something.

Properties

Readability counts, so Python provides a way to avoid writing "getters and setters" when unnecessary.

In Java, it's nearly impossible to make everything public, since changing a class to use getters and setters would require a change of everything that interfaces with it.

Python's properties allow you to make your variable public to begin with, and then write getters and setters only once they are needed to actually check something.

Using Properties

```python
class CameraSensor:
    def __init__(self):
        self.brightness = 10

    def take_picture(self):
        # do something
        return image
```

```python
camera = CameraSensor()
camera.brightness = 40
camera.take_picture()
```

Using Properties

```python
class CameraSensor:
    def __init__(self):
        self._brightness = 10

    def take_picture(self):
        # do something
        return image

    @property
    def brightness(self):
        return self._brightness

    @brightness.setter
    def brightness(self, value):
        if not 0 <= value <= 100:
            raise ValueError
        self._brightness = value
```

```python
camera = CameraSensor()
camera.brightness = 40
camera.take_picture()
```

Decorators

Decorators

`@property` as we just saw is what is called a decorator. Decorators are really just a pretty way to wrap functions using functions that return functions.

Both the following are equivalent:

```
@logging
def foo(bar, baz):
    return bar + baz - 42

# equivalent to...
def foo(bar, baz):
    return bar + baz - 42
foo = logging(foo)
```

Decorators

`@property` as we just saw is what is called a decorator. Decorators are really just a pretty way to wrap functions using functions that return functions.

Both the following are equivalent:

```
@logging
def foo(bar, baz):
    return bar + baz - 42

# equivalent to...
def foo(bar, baz):
    return bar + baz - 42
foo = logging(foo)
```

Defining Decorators

When defining wrapper functions, you should decorate it with `wraps` from `functools`, this will keep attributes about the function.

```
from functools import wraps

def logging(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        print(result)
        return result
    return wrapper
```

Decorators in the Wild: Dynamic Programming

`lru_cache` from `functools` can be a quick way to make a recursive function with a recurrence relation fast. Here's an example:

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci(n):
    if n == 0 or n == 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Decorators in the Wild: Dynamic Programming

`lru_cache` from `functools` can be a quick way to make a recursive function with a recurrence relation fast. Here's an example:

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci(n):
    if n == 0 or n == 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Decorators in the Wild: Welford's Equations

Welford's Equations are a one-pass mean and standard deviation algorithm. One important property is that we won't have to store the results in a list.

Our goal will be to implement a decorator we can use like this:

```python
@Welford
def diceroll(u):
    return int(u * 6) + 1

# call diceroll with some u's in (0, 1)

print(diceroll.mean, diceroll.stdev)
```

Decorators in the Wild: Welford's Equations

Welford's Equations are a one-pass mean and standard deviation algorithm. One important property is that we won't have to store the results in a list.

Our goal will be to implement a decorator we can use like this:

```
@Welford
def diceroll(u):
    return int(u * 6) + 1

# call diceroll with some u's in (0, 1)

print(diceroll.mean, diceroll.stdev)
```

Decorators in the Wild: Implementing Welford

The key here is that we can make callable objects using `__call__`.

```
from functools import update_wrapper
from math import sqrt

class Welford:
    def __init__(self, f):
        self.f = f
        update_wrapper(self, f)
        self.mean = 0
        self.v = 0
        self.trials = 0

    def __call__(self, *args, **kwargs):
        r = self.f(*args, **kwargs)
        self.trials += 1
        d = r - self.mean
        self.v += d**2 * (self.trials - 1)/self.trials
        self.mean += d/self.trials
        return r

    @property
    def stdev(self):
        return sqrt(self.v/self.trials) if self.trials else 0
```

More Decorator Tricks

- Decorators can wrap classes as well as functions. A practical example might be creating a decorator which adds attributes of a class to a database (a `@model` decorator?)
- When multiple decorators are typed, they are applied bottom-up.

More Decorator Tricks

- Decorators can wrap classes as well as functions. A practical example might be creating a decorator which adds attributes of a class to a database (a `@model` decorator?)
- When multiple decorators are typed, they are applied bottom-up.

Generators & Comprehensions

Generator Expressions

Remember the generator function from earlier? Generators can be written inline, these are called **generator expressions**.

```
(x + 4 for x in nums if x % 2 == 0)
```

There's two parts to a generator expression:

1. Performing something for every element with `for...in`.
2. Selecting a subset of elements to operate on with `if`. This part is optional.

The Pythonic Way

Generator expressions are key to the art of Python. Without extensive knowledge of generator expressions, one will forever be a novice Pythonist.

Generator Expressions

Remember the generator function from earlier? Generators can be written inline, these are called **generator expressions**.

```
(x + 4 for x in nums if x % 2 == 0)
```

There's two parts to a generator expression:

1. Performing something for every element with `for...in`.
2. Selecting a subset of elements to operate on with `if`. This part is optional.

The Pythonic Way

Generator expressions are key to the art of Python. Without extensive knowledge of generator expressions, one will forever be a novice Pythonist.

Generator Expressions

Remember the generator function from earlier? Generators can be written inline, these are called **generator expressions**.

```
(x + 4 for x in nums if x % 2 == 0)
```

There's two parts to a generator expression:

1. Performing something for every element with `for...in`.
2. Selecting a subset of elements to operate on with `if`. This part is optional.

The Pythonic Way

Generator expressions are key to the art of Python. Without extensive knowledge of generator expressions, one will forever be a novice Pythonist.

Generator Expressions

Remember the generator function from earlier? Generators can be written inline, these are called **generator expressions**.

```
(x + 4 for x in nums if x % 2 == 0)
```

There's two parts to a generator expression:

1. Performing something for every element with `for...in`.
2. Selecting a subset of elements to operate on with `if`. This part is optional.

The Pythonic Way

Generator expressions are key to the art of Python. Without extensive knowledge of generator expressions, one will forever be a novice Pythonist.

Expression Syntax

```
(expression for expr in sequence1
            if condition1
            for expr2 in sequence2
            if condition2
            for expr3 in sequence3 ...
            if condition3
            for exprN in sequenceN
            if conditionN)
```

Notice the loops are evaluated outside-in.

Applications of Generator Expressions

- Summing ASCII values of a string

  ```
  sum(ord(c) for c in s)
  ```

  Note that the double-parentheses can be omitted.

- File readers

  ```
  reader = (float(line) for line in f)
  while processing_queue:
      process(next(reader))
  ```

- Hash Function pRNGs

  ```
  rng = (hashfunc(x)/MAXHASH for x in count())
  diceroll(next(rng))
  ```

- The possibilities are endless!

Applications of Generator Expressions

- Summing ASCII values of a string

  ```
  sum(ord(c) for c in s)
  ```

  Note that the double-parentheses can be omitted.

- File readers

  ```
  reader = (float(line) for line in f)
  while processing_queue:
      process(next(reader))
  ```

- Hash Function pRNGs

  ```
  rng = (hashfunc(x)/MAXHASH for x in count())
  diceroll(next(rng))
  ```

- The possibilities are endless!

Applications of Generator Expressions

- Summing ASCII values of a string

  ```
  sum(ord(c) for c in s)
  ```

  Note that the double-parentheses can be omitted.

- File readers

  ```
  reader = (float(line) for line in f)
  while processing_queue:
      process(next(reader))
  ```

- Hash Function pRNGs

  ```
  rng = (hashfunc(x)/MAXHASH for x in count())
  diceroll(next(rng))
  ```

- The possibilities are endless!

Applications of Generator Expressions

- Summing ASCII values of a string

  ```
  sum(ord(c) for c in s)
  ```

  Note that the double-parentheses can be omitted.

- File readers

  ```
  reader = (float(line) for line in f)
  while processing_queue:
      process(next(reader))
  ```

- Hash Function pRNGs

  ```
  rng = (hashfunc(x)/MAXHASH for x in count())
  diceroll(next(rng))
  ```

- The possibilities are endless!

List Comprehensions

Building lists in a syntax like generator expressions can be done simply by using square brackets.

```
my_list = [x + 4 for x in nums if x % 2 == 0]
```

Non-comprehensive Alternative

This is an alternative to the **bad programming** edition:

```
my_list = []
for x in nums:
    if x % 2 == 0:
        my_list.append(x)
```

Please avoid writing **absolute garbage** like this to initialize a data structure! It's convoluted and slow!

List Comprehensions

Building lists in a syntax like generator expressions can be done simply by using square brackets.

```
my_list = [x + 4 for x in nums if x % 2 == 0]
```

Non-comprehensive Alternative

This is an alternative to the **bad programming** edition:

```
my_list = []
for x in nums:
    if x % 2 == 0:
        my_list.append(x)
```

Please avoid writing **absolute garbage** like this to initialize a data structure! It's convoluted and slow!

Generic Comprehensions

The same comprehension syntax can be applied to other data structures like so:

```
# Sets
myset = {foo(x, y) for x, y in points}

# Dictionaries
mydict = {point: dist(p) for p in points}

# Tuples
mytup = tuple(foo(x, y) for x, y in points)
```

Functional Programming

Functional Programming

- High-order functions
- We can do a lot in very few lines
- Allow us to mathematically prove our algorithms correct, that's better than any finite amount of unit tests!
- Decorators are a little piece of functional programming
- Generator expressions are also a form of functional programming

Functional Programming

- High-order functions
- We can do a lot in very few lines
- Allow us to mathematically prove our algorithms correct, that's better than any finite amount of unit tests!
- Decorators are a little piece of functional programming
- Generator expressions are also a form of functional programming

Functional Programming

- High-order functions
- We can do a lot in very few lines
- Allow us to mathematically prove our algorithms correct, that's better than any finite amount of unit tests!
- Decorators are a little piece of functional programming
- Generator expressions are also a form of functional programming

Functional Programming

- High-order functions
- We can do a lot in very few lines
- Allow us to mathematically prove our algorithms correct, that's better than any finite amount of unit tests!
- Decorators are a little piece of functional programming
- Generator expressions are also a form of functional programming

Functional Programming

- High-order functions
- We can do a lot in very few lines
- Allow us to mathematically prove our algorithms correct, that's better than any finite amount of unit tests!
- Decorators are a little piece of functional programming
- Generator expressions are also a form of functional programming

min/max

`min`/`max` gets the minimum or maximum value from an iterable, optionally using a key function to select by.

Example:

```
x = min(points, key=lambda p:dist(p, z))
```

The *Bad Programming* Version

```
min_dist = float('inf')
for p in points:
    d = dist(p, z)
    if d < min_dist:
        x = p
```

Don't do this crap!

min/max

`min/max` gets the minimum or maximum value from an iterable, optionally using a key function to select by.

Example:

```
x = min(points, key=lambda p:dist(p, z))
```

The *Bad Programming* Version

```
min_dist = float('inf')
for p in points:
    d = dist(p, z)
    if d < min_dist:
        x = p
```

Don't do this crap!

min/max

`min/max` gets the minimum or maximum value from an iterable, optionally using a key function to select by.

Example:

```
x = min(points, key=lambda p:dist(p, z))
```

The *Bad Programming* Version

```
min_dist = float('inf')
for p in points:
    d = dist(p, z)
    if d < min_dist:
        x = p
```

Don't do this crap!

`zip`

`zip` creates a **iterator** over the *n*th element of each of it's arguments (which are iterables).

Example:

```
for a, b, c in zip(list1, list2, list3):
    # do something
```

Pro Tip: Iterating over the columns of a 2D matrix

```
for col in zip(*M):
    # do something with each column
```

zip

`zip` creates a **iterator** over the *n*th element of each of it's arguments (which are iterables).

Example:

```
for a, b, c in zip(list1, list2, list3):
    # do something
```

Pro Tip: Iterating over the columns of a 2D matrix

```
for col in zip(*M):
    # do something with each column
```

`zip`

`zip` creates a **iterator** over the *n*th element of each of it's arguments (which are iterables).

Example:

```
for a, b, c in zip(list1, list2, list3):
    # do something
```

Pro Tip: Iterating over the columns of a 2D matrix

```
for col in zip(*M):
    # do something with each column
```

Other Functional Things

- `map(func, *iterables)`, which calls `func(*t)` for all `t` in `zip(*iterables)`. Note that `map` is completely unnecessary as the same can be done using generator expressions. Under a few cases, it may be better to use `map` to improve readability.
- `reduce(func, sequence)` which reduces a sequence by calling `func(func(func(a, b), c), ...)`. This is useful for taking the product of a sequence (use `operator.mul`)

Other Functional Things

- `map(func, *iterables)`, which calls `func(*t)` for all `t` in `zip(*iterables)`. Note that `map` is completely unnecessary as the same can be done using generator expressions. Under a few cases, it may be better to use `map` to improve readability.
- `reduce(func, sequence)` which reduces a sequence by calling `func(func(func(a, b), c), ...)`. This is useful for taking the product of a sequence (use `operator.mul`)

Recommended Reading

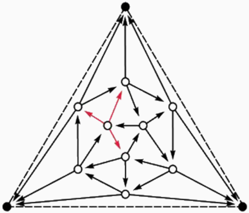
The **Functional Programming HOWTO** page in the Python documentation has some very useful tips for functional programming.

`https://docs.python.org/howto/functional.html`

Useful Libraries

itertools

- Built into Python's standard library
- Features common generator functions
- Features generator functions for iterating over various combinatorics, eg. permutations



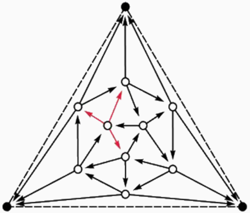
Example: Exhaustive Search Over Permutations

```
def tourlen(tour):
    return sum(dist(a, b) for a, b in zip(tour, tour[1:] + tour[:1]))

def exhaustive(points):
    st = points.pop(0)
    return list(min(((st,) + p for p in permutations(points)), key=tourlen))
```

itertools

- Built into Python's standard library
- Features common generator functions
- Features generator functions for iterating over various combinatorics, eg. permutations



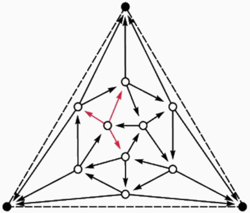
Example: Exhaustive Search Over Permutations

```
def tourlen(tour):
    return sum(dist(a, b) for a, b in zip(tour, tour[1:] + tour[:1]))

def exhaustive(points):
    st = points.pop(0)
    return list(min(((st,) + p for p in permutations(points)), key=tourlen))
```

itertools

- Built into Python's standard library
- Features common generator functions
- Features generator functions for iterating over various combinatorics, eg. permutations



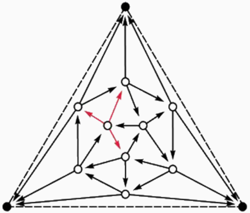
Example: Exhaustive Search Over Permutations

```
def tourlen(tour):
    return sum(dist(a, b) for a, b in zip(tour, tour[1:] + tour[:1]))

def exhaustive(points):
    st = points.pop(0)
    return list(min(((st,) + p for p in permutations(points)), key=tourlen))
```

itertools

- Built into Python's standard library
- Features common generator functions
- Features generator functions for iterating over various combinatorics, eg. permutations



Example: Exhaustive Search Over Permutations

```
def tourlen(tour):
    return sum(dist(a, b) for a, b in zip(tour, tour[1:] + tour[:1]))

def exhaustive(points):
    st = points.pop(0)
    return list(min(((st,) + p for p in permutations(points)), key=tourlen))
```

Requests

- Useful library to do make HTTP requests easy
- Requests is the only *Non-GMO* HTTP library for Python, safe for human consumption.

Behold! The power of Requests!

```
>>> r = requests.get('https://api.github.com/user', auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.encoding
'utf-8'
>>> r.text
u'{"type":"User"...'
>>> r.json()
{u'private_gists': 419, u'total_private_repos': 77, ...}
```

Requests

- Useful library to do make HTTP requests easy
- Requests is the only *Non-GMO* HTTP library for Python, safe for human consumption.

Behold! The power of Requests!

```
>>> r = requests.get('https://api.github.com/user', auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.encoding
'utf-8'
>>> r.text
u'{"type":"User"...'
>>> r.json()
{u'private_gists': 419, u'total_private_repos': 77, ...}
```

Requests

- Useful library to do make HTTP requests easy
- Requests is the only *Non-GMO* HTTP library for Python, safe for human consumption.

Behold! The power of Requests!

```
>>> r = requests.get('https://api.github.com/user', auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.encoding
'utf-8'
>>> r.text
u'{"type":"User"...'
>>> r.json()
{u'private_gists': 419, u'total_private_repos': 77, ...}
```

Bottle: Really Simple Web Framework

- Provides routing and convenient access to data
- Built in HTTP server, or use any WSGI-compatible web server
- Very lightweight, only a couple thousand lines of code

A *Hello, World!* App

```python
from bottle import route, run, template

@route('/hello/<name>')
def index(name):
    return template('<b>Hello {{name}}</b>!', name=name)

run(host='localhost', port=8080)
```

Bottle: Really Simple Web Framework

- Provides routing and convenient access to data
- Built in HTTP server, or use any WSGI-compatible web server
- Very lightweight, only a couple thousand lines of code

A *Hello, World!* App

```
from bottle import route, run, template

@route('/hello/<name>')
def index(name):
    return template('<b>Hello {{name}}</b>!', name=name)

run(host='localhost', port=8080)
```

Bottle: Really Simple Web Framework

- Provides routing and convenient access to data
- Built in HTTP server, or use any WSGI-compatible web server
- Very lightweight, only a couple thousand lines of code

A *Hello, World!* App

```python
from bottle import route, run, template

@route('/hello/<name>')
def index(name):
    return template('<b>Hello {{name}}</b>!', name=name)

run(host='localhost', port=8080)
```

Bottle: Really Simple Web Framework

- Provides routing and convenient access to data
- Built in HTTP server, or use any WSGI-compatible web server
- Very lightweight, only a couple thousand lines of code

A *Hello, World!* App

```python
from bottle import route, run, template

@route('/hello/<name>')
def index(name):
    return template('<b>Hello {{name}}</b>!', name=name)

run(host='localhost', port=8080)
```

Library Suggestions?

Learning Resources

Learning Resources

- The Python documentation is excellent, and includes many tutorials and howtos that may be more readable to a beginner
- My slides[4] from this past summer are online at `https://coding.campinc.com`, these might be better for someone with zero programming experince
- Online courses? I haven't tried any.
- **The Python Cookbook** by *David Beazely* and *Brian K. Jones* is a good book for seasoned Pythonists

[4]These slides aren't complete without someone to teach them.

Learning Resources

- The Python documentation is excellent, and includes many tutorials and howtos that may be more readable to a beginner
- My slides[4] from this past summer are online at `https://coding.campinc.com`, these might be better for someone with zero programming experince
- Online courses? I haven't tried any.
- **The Python Cookbook** by *David Beazely* and *Brian K. Jones* is a good book for seasoned Pythonists

[4]These slides aren't complete without someone to teach them.

Learning Resources

- The Python documentation is excellent, and includes many tutorials and howtos that may be more readable to a beginner
- My slides[4] from this past summer are online at `https://coding.campinc.com`, these might be better for someone with zero programming experince
- Online courses? I haven't tried any.
- **The Python Cookbook** by *David Beazely* and *Brian K. Jones* is a good book for seasoned Pythonists

[4]These slides aren't complete without someone to teach them.

Learning Resources

- The Python documentation is excellent, and includes many tutorials and howtos that may be more readable to a beginner
- My slides[4] from this past summer are online at `https://coding.campinc.com`, these might be better for someone with zero programming experince
- Online courses? I haven't tried any.
- **The Python Cookbook** by *David Beazely* and *Brian K. Jones* is a good book for seasoned Pythonists

[4]These slides aren't complete without someone to teach them.

Questions?

Copyright Notice

Colorado School of Mines
Linux Users Group