10 awesome features of Python that you can't use because you refuse to upgrade to Python 3
10 awesome features of Python that you can't use because you refuse to upgrade to Python 3

or

Turning it up to 11!
10 awesome features of Python that you can't use because you refuse to upgrade to Python 3

or

Turning it up to 11!
Prelude

- Last month (March) APUG: only three people use Python 3 (including me)
- Lots of new features of Python 3.
- Some have been backported to Python 2.7. (like dictionary/set comprehensions or set literals, __future__.print_function)
- But there's more than that.
- New features that you can’t use unless you are in Python 3.
- And it’s more than bytes/unicode...
Feature 0: Matrix Multiplication

- (feature 0 because you can't actually use it yet)
- [PEP 465](http://asmeurer.github.io/python3-presentation/slides.html#1)
- In Python 3.5, you'll be able to replace

```python
>>> a = np.array([[1, 0], [0, 1]])
>>> b = np.array([[4, 1], [2, 2]])
>>> np.dot(a, b)
array([[4, 1],
       [2, 2]])
```

with

```python
>>> a = np.array([[1, 0], [0, 1]])
>>> b = np.array([[4, 1], [2, 2]])
>>> a @ b
array([[4, 1],
       [2, 2]])
```

- Any object can override `__matmul__` to use `@`. 
Feature 1: Advanced unpacking

- You can already do this:

```python
>>> a, b = range(2)
>>> a
0
>>> b
1
```
Feature 1: Advanced unpacking

- You can already do this:

  ```python
  >>> a, b = range(2)
  >>> a
  0
  >>> b
  1
  ```

- Now you can do this:

  ```python
  >>> a, b, *rest = range(10)
  >>> a
  0
  >>> b
  1
  >>> rest
  [2, 3, 4, 5, 6, 7, 8, 9]
  ```
Feature 1: Advanced unpacking

- You can already do this:

```python
>>> a, b = range(2)
>>> a
0
>>> b
1
```

- Now you can do this:

```python
>>> a, b, *rest = range(10)
>>> a
0
>>> b
1
>>> rest
[2, 3, 4, 5, 6, 7, 8, 9]
```

- *rest can go anywhere:

```python
>>> a, *rest, b = range(10)
>>> a
0
>>> b
9
>>> rest
[1, 2, 3, 4, 5, 6, 7, 8]
```

```python
>>> *rest, b = range(10)
>> rest
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> b
9
```
Feature 1: Advanced unpacking

Get the first and last lines of a file

```python
>>> with open("using_python_to_profit") as f:
    ...     first, *__*, last = f.readlines()
>>> first
'Step 1: Use Python 3\n'
>>> last
'Step 10: Profit!\n'
```
Feature 1: Advanced unpacking

Get the first and last lines of a file

```python
>>> with open("using_python_to_profit") as f:
...     first, *___, last = f.readlines()
>>> first
'Step 1: Use Python 3\n'
>>> last
'Step 10: Profit!\n'
```

Refactor your functions

```python
def f(a, b, *args):
    stuff

def f(*args):
    a, b, *args = args
    stuff
```
Feature 2: Keyword only arguments

```python
def f(a, b, *args, option=True):
    ...
```
Feature 2: Keyword only arguments

```python
def f(a, b, *args, option=True):
    ...

    * option comes after *args.
```
Feature 2: Keyword only arguments

```python
def f(a, b, *args, option=True):
    ...
```

- option comes after *args.
- The only way to access it is to explicitly call f(a, b, option=True)
Feature 2: Keyword only arguments

```python
def f(a, b, *args, option=True):
    ...  
```

- option comes after *args.
- The only way to access it is to explicitly call `f(a, b, option=True)`.
- You can write just a * if you don't want to collect *args.

```python
def f(a, b, *, option=True):
    ...  
```
Feature 2: Keyword only arguments

- No more, "Oops, I accidentally passed too many arguments to the function, and one of them was swalled by a keyword argument".

```python
def sum(a, b, biteme=False):
    if biteme:
        shutil.rmtree('/')
    else:
        return a + b

>>> sum(1, 2)
3

>>> sum(1, 2, 3)
```

http://asmeurer.github.io/python3-presentation/slides.html#1

15/72
Feature 2: Keyword only arguments

- No more, "Oops, I accidentally passed too many arguments to the function, and one of them was swalled by a keyword argument".

```python
def sum(a, b, biteme=False):
    if biteme:
        shutil.rmtree('/')
    else:
        return a + b

>>> sum(1, 2)
3

>>> sum(1, 2, 3)
Feature 2: Keyword only arguments

- Instead write

```python
def sum(a, b, *, bitem=False):
    if bitem:
        shutil.rmtree('/')
    else:
        return a + b
```

```python
>>> sum(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sum() takes 2 positional arguments but 3 were given
```
Feature 2: Keyword only arguments

- Instead write

```python
def sum(a, b, *, biteme=False):
    if biteme:
        shutil.rmtree('/')
    else:
        return a + b
```

```python
>>> sum(1, 2, 3)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: sum() takes 2 positional arguments but 3 were given
```
Feature 2: Keyword only arguments

- Or, "I reordered the keyword arguments of a function, but something was implicitly passing in arguments expecting the order"

- Example:

```python
def maxall(iterable, key=None):
    """
    A list of all max items from the iterable
    """
    key = key or (lambda x: x)
    m = max(iterable, key=key)
    return [i for i in iterable if key(i) == key(m)]

>>> maxall(['a', 'ab', 'bc'], len)
['ab', 'bc']
```
Feature 2: Keyword only arguments

- The max builtin supports \texttt{max(a, b, c)}. We should allow that too.

```python
def maxall(*args, key=None):
    """
    A list of all max items from the iterable
    """
    if len(args) == 1:
        iterable = args[0]
    else:
        iterable = args
    key = key or (lambda x: x)
    m = max(iterable, key=key)
    return [i for i in iterable if key(i) == key(m)]
```

- We just broke any code that passed in the key as a second argument without using the keyword.

```python
>>> maxall(['a', 'ab', 'ac'], len)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 10, in maxall
TypeError: unorderable types: builtin_function_or_method() > list()
```

- (Actually in Python 2 it would just return ['a', 'ab', 'ac'], see feature 6).
- By the way, \texttt{max} shows that this is already possible in Python 2, but only if you write your function in C.
- Obviously, we should have used \texttt{maxall(iterable, \*, key=None)} to begin with.
Feature 2: Keyword only arguments

- You can make your APIs "future change proof".
- Stupid example:

```python
def extendto(value, shorter, longer):
    
    """
    Extend list `shorter` to the length of list `longer` with `value`
    """
    if len(shorter) > len(longer):
        raise ValueError('The `shorter` list is longer than the `longer` list')
    a.extend([value]*(len(longer) - len(shorter)))

>>> a = [1, 2]
>>> b = [1, 2, 3, 4, 5]
>>> extendto(10, a, b)
>>> a
[1, 2, 10, 10, 10]
```
Feature 2: Keyword only arguments

- You can make your APIs "future change proof".
- Stupid example:

```python
def extendto(value, shorter, longer):
    """
    Extend list `shorter` to the length of list `longer` with `value`
    
    if len(shorter) > len(longer):
        raise ValueError('The `shorter` list is longer than the `longer` list')
    a.extend([value]*(len(longer) - len(shorter)))
```

```python
>>> a = [1, 2]
>>> b = [1, 2, 3, 4, 5]
>>> extendto(10, a, b)
>>> a
[1, 2, 10, 10, 10]
```

- Hmm, maybe it makes more sense for `longer` to come before `shorter`...
- Too bad, you'll break the code.
Feature 2: Keyword only arguments

- In Python 3, you can use

```python
def extendto(value, *, shorter=None, longer=None):
    """
    Extend list `shorter` to the length of list `longer` with `value`
    """
    if shorter is None or longer is None:
        raise TypeError('`shorter` and `longer` must be specified')
    if len(shorter) > len(longer):
        raise ValueError('The `shorter` list is longer than the `longer` list')
a.extend([value]*(len(longer) - len(shorter)))
```

- Now, `a` and `b` have to be passed in as `extendto(10, shorter=a, longer=b)`.
Feature 2: Keyword only arguments

- In Python 3, you can use

```python
def extendto(value, *, shorter=None, longer=None):
    """
    Extend list `shorter` to the length of list `longer` with `value`
    """
    if shorter is None or longer is None:
        raise TypeError("`shorter` and `longer` must be specified")
    if len(shorter) > len(longer):
        raise ValueError("The `shorter` list is longer than the `longer` list")
    a.extend([value]*(len(longer) - len(shorter)))
```

- Now, a and b have to be passed in as `extendto(10, shorter=a, longer=b)`.
- Or if you prefer, `extendto(10, longer=b, shorter=a)`.
Feature 2: Keyword only arguments

- Add new keyword arguments without breaking API.
- Python 3 did this in the standard library.
Feature 2: Keyword only arguments

- Add new keyword arguments without breaking API.
- Python 3 did this in the standard library.
- For example, functions in os have `follow_symlinks` option.
Feature 2: Keyword only arguments

- Add new keyword arguments without breaking API.
- Python 3 did this in the standard library.
- For example, functions in os have `follow_symlinks` option.
- So you can just use `os.stat(file, follow_symlinks=False)` instead of `os.lstat`. 
Feature 2: Keyword only arguments

- Add new keyword arguments without breaking API.
- Python 3 did this in the standard library.
- For example, functions in `os` have `follow_symlinks` option.
- So you can just use `os.stat(file, follow_symlinks=False)` instead of `os.lstat`.
- In case that sounds more verbose, it lets you do

```python
s = os.stat(file, follow_symlinks=some_condition)
```

instead of

```python
if some_condition:
    s = os.stat(file)
else:
    s = os.lstat(file)
```
Feature 2: Keyword only arguments

- Add new keyword arguments without breaking API.
- Python 3 did this in the standard library.
- For example, functions in os have follow_symlinks option.
- So you can just use os.stat(file, follow_symlinks=False) instead of os.lstat.
- In case that sounds more verbose, it lets you do

```
s = os.stat(file, follow_symlinks=some_condition)
```

instead of

```
if some_condition:
    s = os.stat(file)
else:
    s = os.lstat(file)
```

- But os.stat(file, some_condition) doesn't work.
- Keeps you from thinking it's a two-argument function.
Feature 2: Keyword only arguments

- In Python 2, you have to use **kwargs and do the handling yourself.
Feature 2: Keyword only arguments

- In Python 2, you have to use **kwargs and do the handling yourself.
- Let's of ugly option = kwargs.pop(True) at the top of your functions.
Feature 2: Keyword only arguments

- In Python 2, you have to use **kwargs and do the handling yourself.
- Lots of ugly option = kwargs.pop(True) at the top of your functions.
- No longer self documenting.
Feature 2: Keyword only arguments

- In Python 2, you have to use **kwargs and do the handling yourself.
- Lot's of ugly `option = kwargs.pop(True)` at the top of your functions.
- No longer self documenting.
- If you somehow are writing for a Python 3 only codebase, I highly recommend making all your keyword arguments keyword only, especially keyword arguments that represent "options".
Feature 3: Chained exceptions

- **Situation:** you catch an exception with except, do something, and then raise a different exception.

```python
def mycopy(source, dest):
    try:
        shutil.copy2(source, dest)
    except OSError:  # We don't have permissions. More on this later
        raise NotImplementedError("automatic sudo injection")
```

- **Problem:** You lose the original traceback

```python
>>> mycopy('noway', 'noway2')
>>> mycopy(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in mycopy
NotImplementedError: automatic sudo injection
```
Feature 3: Chained exceptions

- **Situation:** you catch an exception with `except`, do something, and then raise a different exception.

```python
def mycopy(source, dest):
    try:
        shutil.copy2(source, dest)
    except OSError:  # We don't have permissions. More on this later
        raise NotImplementedError("automatic sudo injection")
```

- **Problem:** You lose the original traceback

```python
>>> mycopy('noway', 'noway2')
>>> mycopy(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in mycopy
NotImplementedError: automatic sudo injection
```

- What happened with the `OSError`?
Feature 3: Chained exceptions

- Python 3 shows you the whole chain of exceptions:

```python
mycopy('noway', 'noway2')
Traceback (most recent call last):
File "<stdin>", line 3, in mycopy
File "~/anaconda3/lib/python3.3/shutil.py", line 243, in copy2
    copyfile(src, dst, follow_symlinks=follow_symlinks)
File "~/anaconda3/lib/python3.3/shutil.py", line 109, in copyfile
    with open(src, 'rb') as fsrc:
PermissionError: [Errno 13] Permission denied: 'noway'
During handling of the above exception, another exception occurred:

Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 5, in mycopy
NotImplementedError: automatic sudo injection
```
Feature 3: Chained exceptions

- Python 3 shows you the whole chain of exceptions:

```python
mycopy('noway', 'noway2')
Traceback (most recent call last):
File "<stdin>", line 3, in mycopy
    copyfile(src, dst, follow_symlinks=follow_symlinks)
File "/Users/aaronmeurer/anaconda3/lib/python3.6/shutil.py", line 109, in copyfile
    with open(src, 'rb') as fsrc:
PermissionError: [Errno 13] Permission denied: 'noway'
```

During handling of the above exception, another exception occurred:

```python
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 5, in mycopy
NotImplementedError: automatic sudo injection
```

- You can also do this manually using `raise from`

```python
raise exception from e
```

```python
>>> raise NotImplementedError from OSError
OSError
The above exception was the direct cause of the following exception:

Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NotImplementedError
```
Feature 4: Fine grained OSError subclasses

- The code I just showed you is wrong.
- It catches OSError and assumes it is a permission error.
- But OSError can be a lot of things (file not found, is a directory, is not a directory, broken pipe, ...)
- You really have to do

```python
import errno
def mycopy(source, dest):
    try:
        shutil.copy2(source, dest)
    except OSError as e:
        if e.errno in [errno.EPERM, errno.EACCES]:
            raise NotImplementedError("automatic sudo injection")
        else:
            raise
```
Feature 4: Fine grained OSError subclasses

- The code I just showed you is wrong.
- It catches OSError and assumes it is a permission error.
- But OSError can be a lot of things (file not found, is a directory, is not a directory, broken pipe, ...)
- You really have to do

```python
import errno
def mycopy(source, dest):
    try:
        shutil.copy2(source, dest)
    except OSError as e:
        if e.errno in [errno.EPERM, errno.EACCES]:
            raise NotImplementedError("automatic sudo injection")
        else:
            raise
```

- Wow. That sucks.
Feature 4: Fine grained OSError subclasses

- The code I just showed you is wrong.
- It catches OSError and assumes it is a permission error.
- But OSError can be a lot of things (file not found, is a directory, is not a directory, broken pipe, ...)
- You really have to do

```python
import errno

def mycopy(source, dest):
    try:
        shutil.copy2(source, dest)
    except OSError as e:
        if e.errno in [errno.EPERM, errno.EACCES]:
            raise NotImplementedError("automatic sudo injection")
        else:
            raise
```

- Wow. That sucks.
Feature 4: Fine grained OSError subclasses

- Python 3 fixes this by adding a ton of new exceptions.
- You can just do

```python
def mycopy(source, dest):
    try:
        shutil.copy2(source, dest)
    except PermissionError:
        raise NotImplementedError("automatic sudo injection")
```

- (Don't worry, PermissionError subclasses from OSError and still has .errno. Old code will still work).
Feature 5: Everything is an iterator

- This is the hardest one to sell.
- Iterators exist in Python 2 as well.
- But you have to use them. Don't write `range` or `zip` or `dict.values` or ....
Feature 5: Everything is an iterator

- If you do...
Feature 5: Everything is an iterator

- If you do...

```python
def naivesum(N):
    """
    Naively sum the first N integers
    """
    A = 0
    for i in range(N + 1):
        A += i
    return A
```
Feature 5: Everything is an iterator

- If you do...

```python
def naivesum(N):
    """
    Naively sum the first N integers
    """
    A = 0
    for i in range(N + 1):
        A += i
    return A
```

```
In [3]: timeit naivesum(1000000)
10 loops, best of 3: 61.4 ms per loop
```
Feature 5: Everything is an iterator

- If you do...

```python
def naivesum(N):
    """
    Naively sum the first N integers
    """
    A = 0
    for i in range(N + 1):
        A += i
    return A
```

In [3]: timeit naivesum(1000000)
10 loops, best of 3: 61.4 ms per loop

In [4]: timeit naivesum(10000000)
1 loops, best of 3: 622 ms per loop
Feature 5: Everything is an iterator

- If you do...

```python
def naive_sum(N):
    """
    Naively sum the first N integers
    """
    A = 0
    for i in range(N + 1):
        A += i
    return A
```

In [3]: timeit naive_sum(1000000)
10 loops, best of 3: 61.4 ms per loop

In [4]: timeit naive_sum(10000000)
1 loops, best of 3: 622 ms per loop

In [5]: timeit naive_sum(100000000)
Feature 5: Everything is an iterator

- If you do...

```python
def naivesum(N):
    """
    Naively sum the first N integers
    """
    A = 0
    for i in range(N + 1):
        A += i
    return A
```

```
In [3]: %timeit naivesum(1000000)
10 loops, best of 3: 61.4 ms per loop

In [4]: %timeit naivesum(10000000)
1 loops, best of 3: 622 ms per loop

In [5]: %timeit naivesum(100000000)
```

Feature 5: Everything is an iterator

IF YOU USE RANGE() IN PYTHON 2
YOU'RE GONNA HAVE A BAD TIME
Feature 5: Everything is an iterator

- Instead write some variant (xrange, itertools.izip, dict.itervalues, ...).
- Inconsistent API anyone?
Feature 5: Everything is an iterator

- In Python 3, range, zip, map, dict.values, etc. are all iterators.
- If you want a list, just wrap the result with list.
- Explicit is better than implicit.
- Harder to write code that accidentally uses too much memory, because the input was bigger than you expected.
Feature 6: No more comparison of everything to everything

- In Python 2, you can do

```python
>>> max(['one', 2]) # One *is* the lonliest number
'one'
```
Feature 6: No more comparison of everything to everything

- In Python 2, you can do

```
>>> max(['one', 2])  # One *is* the lonliest number
'one'
```

- Hurray. I just disproved math!

```
\[
\begin{align*}
a &= b \\
a + a &= a + b \\
2a &= a + b \\
2a - 2b &= a + b - 2b \\
2 (a - b) &= a + b - 2b \\
2 (a - b) &= a - b \\
2 &= 1
\end{align*}
\]
```

Your God

Where is he now?
Feature 6: No more comparison of everything to everything

- It's because in Python 2, you can < compare anything to anything.

```python
>>> 'abc' > 123
True
>>> None > all
False
```
Feature 6: No more comparison of everything to everything

- It's because in Python 2, you can `<` compare anything to anything.
  ```python
  >>> 'abc' > 123
  True
  >>> None > all
  False
  ```

- In Python 3, you can’t do this:
  ```python
  >>> 'one' > 2
  Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  TypeError: unorderable types: str() > int()
  ```

- This avoids subtle bugs, e.g., from not coercing all types from int to str or visa versa.
- Especially when you use `>` implicitly, like with `max` or `sorted`.

- In Python 2:
  ```python
  >>> sorted(['1', 2, '3'])
  [2, '1', '3']
  ```
Feature 7: yield from

- Pretty great if you use generators
- Instead of writing

```python
for i in gen():
    yield i
```

Just write

```python
yield from gen()
```

- Easily refactor generators into subgenerators.
Feature 7: yield from

- Makes it easier to turn everything into a generator. See "Feature 5: Everything is an iterator" above for why you should do this.

- Instead of accumulating a list, just yield or yield from.

**Bad**

```python
def dup(n):
    A = []
    for i in range(n):
        A.extend([i, i])
    return A
```

**Good**

```python
def dup(n):
    for i in range(n):
        yield i
    yield i
```

**Better**

```python
def dup(n):
    for i in range(n):
        yield from [i, i]
```
Feature 7: yield from

In case you don't know, generators are awesome because:

- Only one value is computed at a time. Low memory impact (see range example above).
- Can break in the middle. Don't have to compute everything just to find out you needed none of it. Compute just what you need. If you often don't need it all, you can gain a lot of performance here.
- If you need a list (e.g., for slicing), just call list() on the generator.
- Function state is "saved" between yields.
- This leads to interesting possibilities, a la. coroutines...
Feature 8: asyncio

- Uses new coroutines features and saved state of generators to do asynchronous IO.

```python
# Taken from Guido's slides from “Tulip: Async I/O for Python 3” by Guido
# van Rossum, at LinkedIn, Mountain View, Jan 23, 2014
@coroutine
def fetch(host, port):
    r, w = yield from open_connection(host, port)
    w.write(b'GET /HTTP/1.0\r\n\r
 ')
    while (yield from r.readline()).decode('latin-1').strip():
        pass
    body = yield from r.read()
    return body

@coroutine
def start():
    data = yield from fetch('python.org', 80)
    print(data.decode('utf-8'))
```
Feature 8: asyncio

- Uses new coroutines features and saved state of generators to do asynchronous IO.

```python
# Taken from Guido's slides from “Tulip: Async I/O for Python 3” by Guido
# van Rossum, at LinkedIn, Mountain View, Jan 23, 2014
@coroutine
def fetch(host, port):
    r,w = yield from open_connection(host,port)
    w.write(b'GET /HTTP/1.0\n\n\n ')
    while (yield from r.readline()).decode('latin-1').strip():
        pass
    body=yield from r.read()
    return body

@coroutine
def start():
    data = yield from fetch('python.org', 80)
    print(data.decode('utf-8'))
```

- Not going to lie to you. I still don't get this.
Feature 8: asyncio

- Uses new coroutines features and saved state of generators to do asynchronous IO.

```python
# Taken from Guido's slides from “Tulip: Async I/O for Python 3” by Guido
# van Rossum, at LinkedIn, Mountain View, Jan 23, 2014
@coroutine
def fetch(host, port):
    r, w = yield from open_connection(host, port)
    w.write(b'GET /HTTP/1.0\r\n\r\n ')
    while (yield from r.readline()).decode('latin-1').strip():
        pass
    body = yield from r.read()
    return body

@coroutine
def start():
    data = yield from fetch('python.org', 80)
    print(data.decode('utf-8'))
```

- Not going to lie to you. I still don't get this.

- It's OK, though. Even David Beazley had a hard time with it:

```
David Beazley
@dabeaz

Trying to decide if I have enough spare brain cells to understand the Python3.4 asyncio module.
1:59 PM - 2 Mar 2014
1 RETWEET 3 FAVORITES
```

http://asmeurer.github.io/python3-presentation/slides.html#1
Feature 9: Standard library additions

faulthandler

- Display (limited) tracebacks, even when Python dies the hard way.
- Won't work with kill -9, but does work with, e.g., segfaults.

```python
import faulthandler
faulthandler.enable()

def killme():
    # Taken from http://nbviewer.ipython.org/github/ipython/ipython/blob/1.x/examples/notebooks/Part2
    import sys
    from ctypes import CDLL
    # This will crash a Linux or Mac system; equivalent calls can be made on
    # Windows
    dll = '{dylib} if sys.platform == 'darwin' else {so.6}
    libc = CDLL("{libc.%s}" % dll)
    libc.time(-1) # BOOM!!

    killme()
```

```text
$python test.py
Fatal Python error: Segmentation fault

Current thread 0x000007fff81b6310:
File "test.py", line 11 in killme
File "test.py", line 13 in <module>
Segmentation fault: 11
```

- Or kill -6 (SIGABRT)
- Can also enable with python -X faulthandler
Feature 9: Standard library additions

ipaddress

- Exactly that. IP addresses.

```python
>>> ipaddress.ip_address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.ip_address('2001:db8::')
IPv6Address('2001:db8::')
```

- Just another thing you don't want to roll yourself.
Feature 9: Standard library additions

functools.lru_cache

- A LRU cache decorator for your functions.
- From docs.

```python
@lru_cache(maxsize=32)
def get_pep(num):
    'Retrieve text of a Python Enhancement Proposal'
    resource = 'http://www.python.org/dev/peps/pep-%04d/' % num
    try:
        with urllib.request.urlopen(resource) as s:
            return s.read()
    except urllib.error.HTTPError:
        return 'Not Found'

>>> for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991:
...    pep = get_pep(n)
...    print(n, len(pep))

>>> get_pep.cache_info()
CacheInfo(hits=3, misses=0, maxsize=32, currsize=0)
```
Feature 9: Standard library additions

enum

- Finally, an enumerated type in the standard library.
- Python 3.4 only.

```python
>>> from enum import Enum
>>> class Color(Enum):
>>>     red = 1
>>>     green = 2
>>>     blue = 3
... 

- Uses some magic that is only possible in Python 3 (due to metaclass changes):

```python
>>> class Shape(Enum):
>>>     square = 2
>>>     square = 3
... 
Traceback (most recent call last):
... 
TypeError: Attempted to reuse key: 'square'
```
Feature 10: Fun

Unicode variable names

```python
>>> résumé = "knows Python"
>>> π = math.pi
```
Feature 10: Fun

Unicode variable names

```python
>>> résumé = "knows Python"
>>> π = math.pi
```

- Sorry, letter-like characters only.
- ⚠️ "beer" does not work.
Feature 10: Fun

Unicode variable names

```python
>>> résumé = "knows Python"
>>> π = math.pi
```

- Sorry, letter-like characters only.
- ☹ = "beer" does not work.

Function annotations

```python
def f(a: stuff, b: stuff = 2) -> result:
    ...
```

- Annotations can be arbitrary Python objects.
- Python doesn't do anything with the annotations other than put them in an __annotations__ dictionary.

```python
>>> def f(x: int) -> float:
    ...
    pass
    ...
>>> f.__annotations__
{'return': <class 'float'>, 'x': <class 'int'>}
```

- But it leaves open the possibility for library authors to do fun things.
- Example, IPython 2.0 widgets.
- Run IPython notebook (in Python 3) from IPython git checkout and open
  http://127.0.0.1:8888/notebooks/examples/Interactive%20Widgets/Image%20Processing.ipynb
Feature 11: Unicode and bytes

- In Python 2, `str` acts like bytes of data.
- There is also `unicode` type to represent Unicode strings.
Feature 11: Unicode and bytes

- In Python 2, `str` acts like bytes of data.
- There is also `unicode` type to represent Unicode strings.
- In Python 3, `str` is a `string`.
- `bytes` are bytes.
- There is no `unicode. str` strings are Unicode.
Discuss
Slides were made with [http://remarkjs.com/](http://remarkjs.com/).

All images have been blatanly stolen from the internet.

Source for slides can be found at [https://github.com/asmeurer/python3-presentation](https://github.com/asmeurer/python3-presentation).

I am Aaron Meurer (@asmeurer).

I gave this presentation on April 9, 2014 at [APUG](http://apug.com). If you are in Austin, TX and you enjoy Python, you should come to APUG!