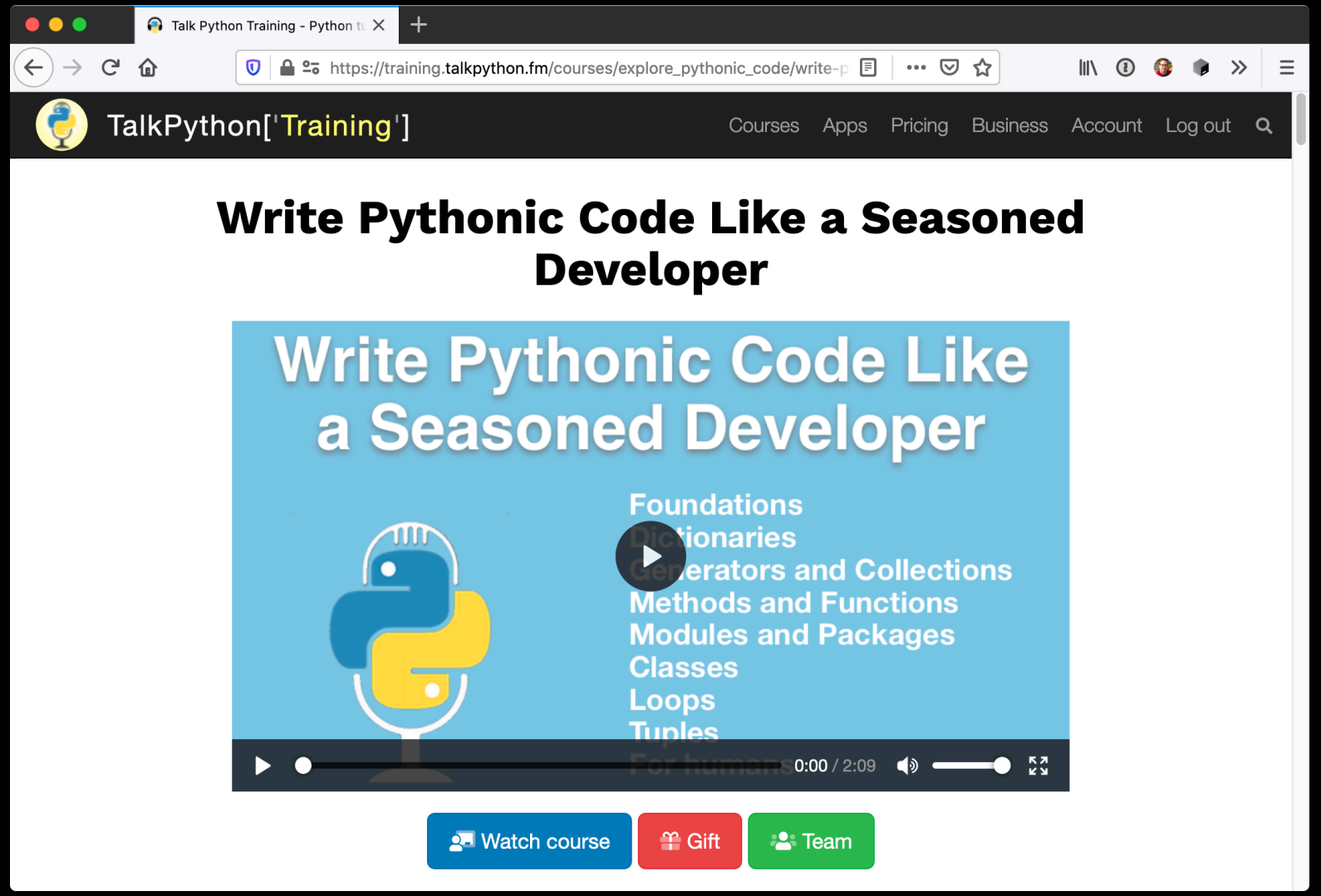
26 Pythonic Code Tips and Tricks

Pythonic tips from the video course

Michael Kennedy
@mkennedy

Take the full video course at Talk Python



https://talkpython.fm/pythonic

What is Pythonic code?

The idea of writing idiomatic code that is most aligned with the language features and ideals is a key concept in Python. We call this idiomatic code **Pythonic***.*

Truthiness

```
False  # false is false
[]     # empty lists / arrays are false
{}     # empty dictionaries are false
""     # empty strings are false
0      # zero ints are false
0.0    # zero floats are false
None   # None / null / nil pointers are false
Custom # Custom __bool__ / __nonzero__

# Everything else is true!
```

Testing for None

```
accounts = find_accounts(search_text)

if accounts is not None:
    # list accounts...
```

Comparisons to singletons use **is**.

Negations with is are "is not" rather than "not account is None"

Multiple tests against a single variable

```
class Moves(Enum): West=1; East=2; ...

if m in {Moves.North, Moves.South, Moves.West, Moves.East}:
    print("That's a direct move.")
else:
    print("That's a diagonal move.")
```

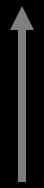

Use `in set()` to test single value against multiple conditions.

String formatting

```
name = 'Michael'
age = 43

# Pythonic
print("Hi, I'm {} and I'm {} years old.".format(name, age))
# Prints: Hi, I'm Michael and I'm 43 years old.

print("Hi, I'm {1} years old and my name is {0}, yeah {1} years.".format(name, age))
# Prints: Hi, I'm 43 years old and my name is Michael, yeah 43 years.

# If you have a dictionary then this is still better:
data = {'day': 'Saturday', 'office': 'Home office', 'other': 'UNUSED'}
print("On {day} I was working in my {office}!".format(**data))
# Prints: On Saturday I was working in my Home office!

# Finally, in Python 3.6, you can:
print(f"On {day} I was working in my {office}!")
# Prints: On Saturday I was working in my Home office!
```

Send an exit code

```
confirm = input("Are you sure you want to format drive C: [yes, NO]? ")

if not confirm or confirm.lower() != 'yes':
    print("Format cancelled!")
     sys.exit(1)

format_drive()
print("Format completed successful.")

sys.exit(0)
```



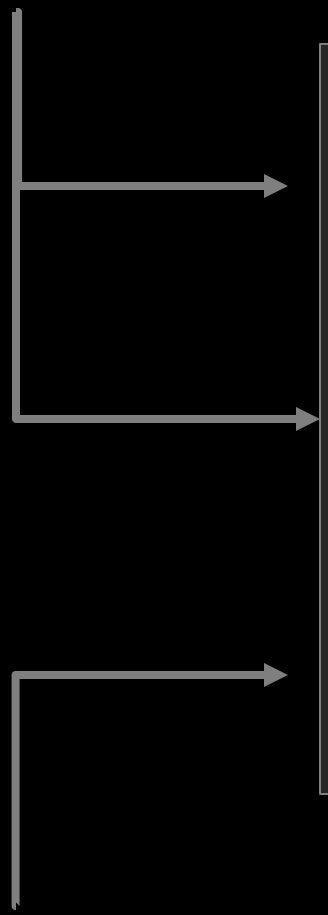
External processes can now observe the success or failure states.

Flat is better than nested

```
def download():
    print("Let's try to download a file")

    if not s.check_download_url():
        print("Bad url")
        return
    if not s.check_network():
        print("No network")
        return
    if not s.check_dns():
        print("No DNS")
        return
    if not s.check_access_allowed():
        print("No access")
        return

    print("Sweet, we can download ...")
```

PEP 8: Imports

Imports should be one line per package, almost always avoid wildcards



```
# BAD
import sys, os, multiprocessing
from statistics import *

# GOOD
import sys
import os
import multiprocessing

# ALSO GOOD
from os import path, chmod, chown
```

Multiple symbols from one module are fine however.

PEP 8: Naming Conventions

Modules have short, all-lowercase names.

Constants are UPPERCASE

Classes have CapWord names.

Variables and args are lowercase

Functions are lowercase

Exceptions should end in Error

```
# module data_access

# constant, not changed at runtime
TRANSCATION_MODE = 'serializable'

class CustomerRepository:

    def __init__(self):
        self.db = create_db()
        self.mode = 'dev'

    def create_customer(self, name, email):
        # ...

class RepositoryError(Exception): pass
```

Merging dictionaries

```
route = {'id': 271, 'title': 'Fast apps'}
query = {'id': 1, 'render_fast': True}
post = {'email': 'j@j.com', 'name': 'Jeff'}

m1 = {**query, **post, **route}
```

Py3

Adding iteration to custom types

```
class ShoppingCart:
    def __init__(self):
        self.items = []

    def add_item(self, it):
        self.items.append(it)

    def __iter__(self):
        for it in sorted(self.items, key=lambda i: i.price):
            yield it

cart = ShoppingCart()
cart.add_item(CartItem("guitar", 799))
cart.add_item(CartItem("cd", 19))
cart.add_item(CartItem("iPhone", 699))

for item in cart:
    print('{} for ${}'.format(item.name, item.price))
```

Testing for containment

```
nums_dict = {1: "one", 2: "two", 3:"three", 5: "five"}

word_version = nums_dict[2] # could KeyError

if 2 in nums_dict:
    word_version = nums_dict[2] # safe!
```

On demand computation with yield

```
def fibonacci_generator():
    current, nxt = 0, 1
    while True:
        current, nxt = nxt, nxt + current
        yield current
```

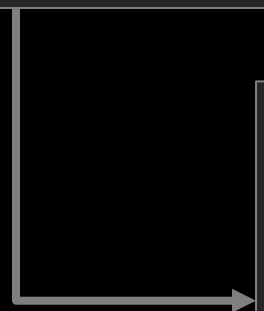
Recursive yields made easy

```
def get_files(folder):
    for item in os.listdir(folder):
        full_item = os.path.join(folder, item)

      if os.path.isfile(full_item):
            yield full_item
       elif os.path.isdir(full_item):
            yield from get_files(full_item)
```

Py3

Inline generators via expressions

```
# direct loop style
high_measurements = []
for m in measurements:
    if m.value >= 70:
        high_measurements.append(m.value)
```



```
# generator style
high_measurements = (
    m.value
    for m in measurements
    if m.value >= 70
)
```

Lambda expressions

```
def find_special_numbers(special_selector, limit=10):
    found = []
    n = 0
    while len(found) < limit:
        if special_selector(n):
            found.append(n)
        n += 1
    return found

find_special_numbers(lambda x: x % 6 == 0)
```


Lambda expressions are small, inline methods.

Avoid return values for error handling

EAFP (easier to ask forgiveness than permission) style

```
def get_latest_file():
    try:
        data = s.download_file()
        save_file('latest.png', data)
        print('Successfully downloaded latest.png')
    except ConnectionError as ce:
        print("Problem with network: {}".format(ce))
    except Exception as x:
        print("Error: {}".format(x))
```

Exception handling catches all the cases and
allows for specific error handling.

Passing variable number of arguments

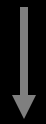
```
def biggest(x, y):
    if x > y:
        return x
    else:
        return y

biggest(1, 7) # 7
```

*NAME signals variable length arguments [0, ...)
args is the conventional name for this argument

```
def biggest(x, *args):  # type(args) = tuple
    b = x
    for y in args:
        if y > b:
            b = y

    return b

biggest_var(1, 7, 11, 99, 5, -2) # 99
```

Mutable values as method defaults

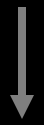
Default value is created once per process.



```
def add_items_bad(name, times=1, lst=[]):
    for _ in range(0, times):
        lst.append(name)
    return lst

list_1 = add_items_bad("a", 3)    # list_1 = [a,a,a]
add_items_bad("b", 2, list_1)     # list_1 = [a,a,a,b,b]

list_2 = add_items_bad("n", 3)    # list_2 = [a,a,a,b,b,n,n,n]
id(list_1) == id(list_2)          # True
```

Mutable values as method defaults

List value is created as needed.



```
def add_items_bad(name, times=1, lst=None):
    if lst is None:
        lst = []

    for _ in range(0, times):
        lst.append(name)
    return lst

list_1 = add_items_bad("a", 3)    # list_1 = [a,a,a]
add_items_bad("b", 2, list_1)     # list_1 = [a,a,a,b,b]

list_2 = add_items_bad("n", 3)    # list_2 = [n,n,n]
id(list_1) == id(list_2)          # False
```

Using __main__

Only execute code (rather than method definitions, etc.) when called directly

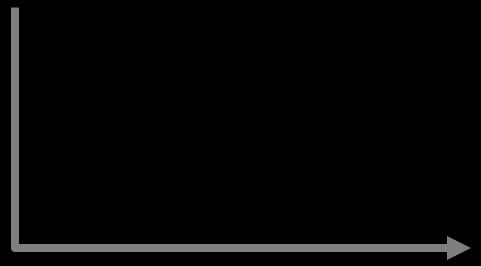
```
# program.py
import support

if __name__ == '__main__':
    print("The variable is {}".format(support.var))
    # prints: The variable is A variable
```

Virtual environments

```
> pip3 install virtualenv

> python3 -m virtualenv ./local_env
Using base prefix
'/Library/Frameworks/Python.framework/Versions/3.5'
New python executable in ./local_env/bin/python3
Also creating executable in ./local_env/bin/python
Installing setuptools, pip...done.

> cd local_env/bin/

> . activate

(local_env) > pip list
pip (8.0.2)
setuptools (19.6.2)
wheel (0.26.0)

(local_env) > which python
/Users/mkennedy/python/environments/local_env/bin/python
```

Defining fields on classes

Define fields in __init__ only.



Almost never create new fields outside of __init__.

```
class NotSoPythonicPet:
    def __init__(self, name, age):
        self.age = age
        self.name = name

    def get_name(self):
        self.name_checked = True
        return self.name

    def get_age(self):
        return self.age
```

Encapsulation and data hiding

```
class PetSnake:
    def __init__(self, name, age):
        self.__age = age
        self.__name = name
        self._protected = True

    @property
    def name(self):
        return self.__name

    @property
    def age(self):
        return self.__age
```

Double _'s mean private.

Single _'s mean protected.

```
py = PetSnake("Slide", 6)
a = py.age # a is 6
protected_v = py._protected # works with warning
direct_a = py.__age # crash:
# AttributeError: 'PetSnake' object has no attribute '__age'
```

Properties (Pythonic)

```
class PetSnake:
    def __init__(self, name, age):
        self.__age = age
        self.__name = name

    @property
    def name(self):
        return self.__name

    @property
    def age(self):
        return self.__age
```

```
py = PetSnake("Slide", 6)

print("She is named {} and {} years old."
    .format(py.name, py.age
))
# prints: She is named Slide and 6 years old.
```

Wait, is there a numerical for loop? [2]

More often, you need the index and the value.
Use enumerate for this.

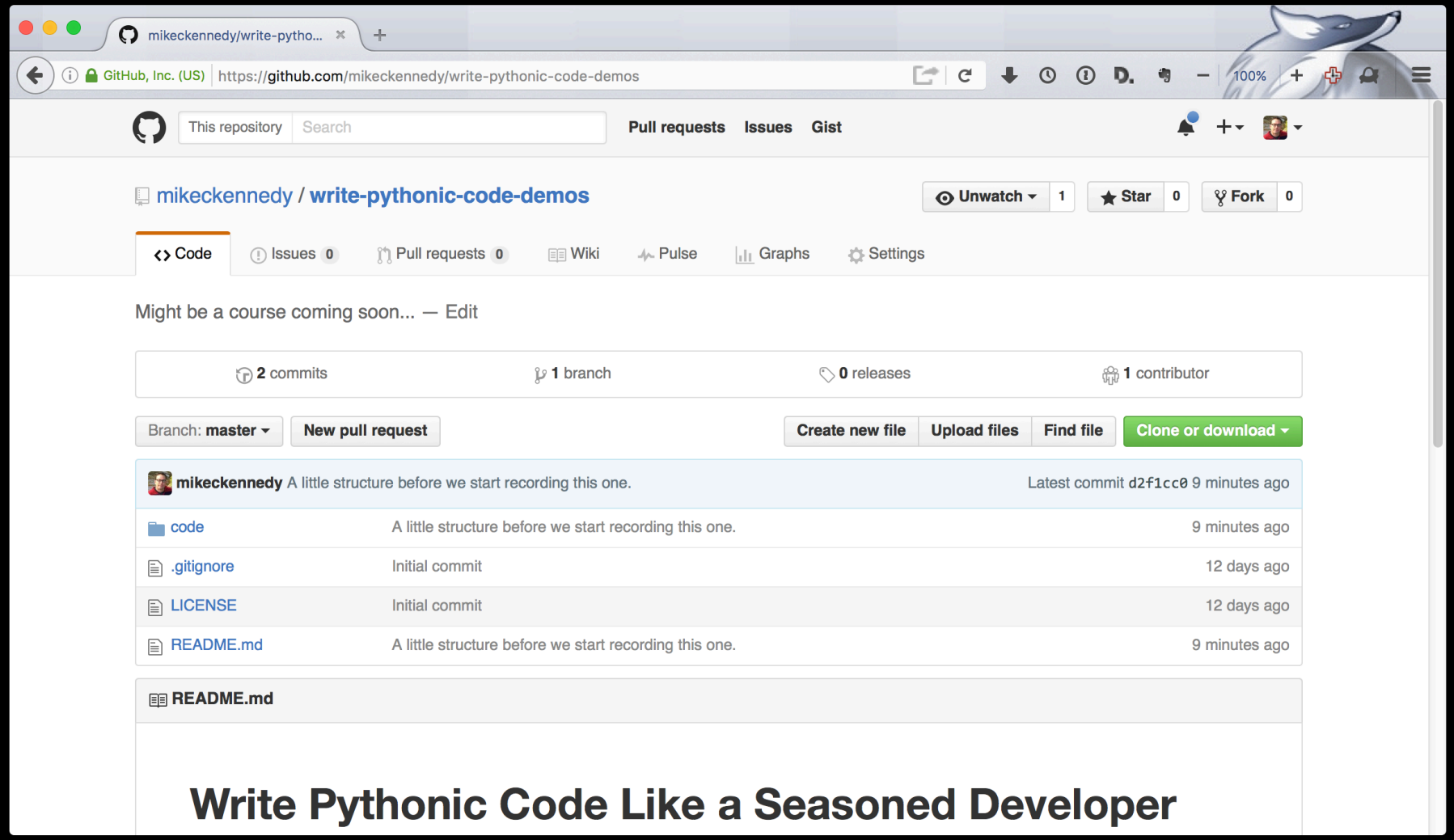
```
data = [1, 7, 11]

for idx, val in enumerate(data):
    print(" {} --> {}".format(idx, val))

# Prints: 0 --> 1, 1 --> 7, 2 --> 11
```

Named Tuples

named tuples are clearer and explicitly named

```
Measurement = collections.namedtuple(
    'Measurement',
    'temp, lat, long, quality')

m = Measurement(22.5, 44.234, 19.02, 'strong')

temp = m[0]
temp = m.temp
quality = m.quality

print(m)
# Measurement(temp=22.5, lat=44.234,
#             long=19.02, quality='strong')
```

Getting the source code



https://github.com/mikeckennedy/write-pythonic-code-demos