Duck Typing and Asking Forgiveness, Not Permission(EAFP)

On many websites I often see comments that code isn't pythonic, or that there is a more pythonic way to achieve the same goal.

What does pythonic mean in this context? For example, why is

while i < someValue:

do\_something(list[i])

i += 1

not pythonic while

for x in list:

doSomething(x)

is pythonic?

Answer- \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Exploiting the features of the Python language to produce code that is clear, concise and maintainable.

Pythonic means code that doesn't just get the syntax right but that follows the conventions of the Python community and uses the language in the way it is intended to be used.

This is maybe easiest to explain by negative example, as in the linked article from the other answers. Examples of unpythonic code often come from users of other languages, who instead of learning a Python programming patterns such as list comprehensions or generator expressions, attempt to crowbar in patterns more commonly used in C or java. Loops are particularly common examples of this.

For example in Java I might use

for i in (i; i < items.length ; i++)

{

n = items[i];

... now do something

}

In Python we can try and replicate this using while loops but it would be cleaner to use

for i in items:

i.perform\_action()

Or, even a generator expression

(i.some\_attribute for i in items)

So essentially when someone says something is unpythonic, they are saying that the code could be re-written in a way that is a better fit for pythons coding style.

Typing import this at the command line gives a summary of Python principles. Less well known is that the source code for import this is decidedly, and by design, unpythonic! Take a look at it for an example of what not to do.

# Duck Typing and Easier to ask forgiveness than permission (EAFP)

class Duck:

def quack(self):

print('Quack, quack')

def fly(self):

print('Flap, Flap!')

class Person:

def quack(self):

print("I'm Quacking Like a Duck!")

def fly(self):

print("I'm Flapping my Arms!")

def quack\_and\_fly(thing):

pass

# Not Duck-Typed (Non-Pythonic)

if isinstance(thing, Duck):

thing.quack()

thing.fly()

else:

print('This has to be a Duck!')

d = Duck()

quack\_and\_fly(d)

# Quack, quack

# Flap, Flap!

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

. . .

def quack\_and\_fly(thing):

# LBYL (Non-Pythonic)

if hasattr(thing, 'quack'): **# can we do this (Asking Permission)**

if callable(thing.quack):  **# can we do this (Asking Permission)**

thing.quack()

if hasattr(thing, 'fly'):  **# can we do this (Asking Permission)**

if callable(thing.fly): **# can we do this (Asking Permission)**

thing.fly()

d = Duck()

quack\_and\_fly(d)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

"""

Duck Typing and Easier to ask forgiveness than permission(EAFP)

- Ducking Typing and EAFP are two different things but closely realted

- Duck Typing: means if an object walks like a duck and quacks like a duck

-  then it is a duck that means we simply don't care what type of object we're

-   working with. We only care if our object can do what we ask it to do.

 """

def quack\_and\_fly(thing):

    pass

    # Not Duck-Typed (Non-Pythonic): because it cares what kind of object it is.

    if isinstance(thing, Duck):  # here it cares what kind of object it is

        # ... it cares whether "thing" is an object of Duck or not

        thing.quack()

        thing.fly()

    else:

        print('This has to be a Duck!')

# but if we want to write it in Duck Typed (Pythonic) approach then we

# write it as below

def quack\_and\_fly(thing):

    pass

    # look, we don't care what kind of object it is. Namely, no                  # checking whether it is what kind of object or instance, but is it safe to

    # pass or permit any object? here is when EAFP comes to rule: see next egs

    thing.quack()

    thing.fly()

    print()

def quack\_and\_fly(thing):

    pass

    try:

        thing.quack()

        thing.fly()

    except AttributeError as e:

        print(e)

    print()

. . .

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def quack\_and\_fly(thing):

**# EAFP (Pythonic)- means try to do something rather than asking, if it**

**# works then great but if it doesn’t then handle the error.**

try:

thing.quack()

thing.fly()

thing.bark()

except AttributeError as e:

print(e)

d = Duck()

quack\_and\_fly(d)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

person = {'name': 'Jess', 'age': 23, 'job': 'Programmer'}

**# Non Pythonic Approach - LBYL (Look Before You Leave)**

if 'name' in person and 'age' in person and 'job' in person:

print("I'm {name} and I am {age} years old who works as {job} in

Google.".format(\*\*person))

else:

print('Missing some keys.')

**# Pythonic Approach**

try:

print("I'm {name} and I am {age} years old.

I works as {job} in Google.".format(\*\*person))

except KeyError as e:

print("Missing {e} key".format(e))

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

import os **# Non Pythonic Approach**

my\_file = '/tmp/test.txt'

if os.access(my\_file, os.R\_OK): **# os.R\_OK to test the readability of file**

with open(my\_file) as f:

print(f.read())

else:

print('File can not be accessed')

try: **# Pythonic Approach**

f = open(my\_file)

except IOError as e:

print('File can not be accessed')

else:

with f:

print(f.read())