Duck Typing, Scope, and Investigative Functions in Python

by **Adrian Tam** on February 16, 2022 in **Python for Machine Learning**

Tweet Tweet Share Share

Last Updated on June 21, 2022

Python is a duck typing language. It means the data types of variables can change as long as the syntax is compatible. Python is also a dynamic programming language. Meaning we can change the program while it runs, including defining new functions and the scope of the name resolution. These give us not only a new paradigm in writing Python code but also a new set of tools for debugging. In the following, we will see what we can do in Python that cannot be done in many other languages.

After finishing this tutorial, you will know:

- How Python manages the variables you define
- How Python code uses a variable and why we don't need to define its type like in C or Java

Kick-start your project with my new book Python for Machine Learning, including *step-by-step tutorials* and the *Python source code* files for all examples.

Let's get started.

Duck typing, scope, and investigative functions in Python. Photo by Julissa Helmuth. Some rights reserved

Overview

This tutorial is in three parts; they are

- Duck typing in programming languages
- Scopes and name space in Python
- Investigating the type and scope

Duck Typing in Programming Languages

Duck typing is a feature of some modern programming languages that allow data types to be dynamic.

> *A programming style which does not look at an object’s type to determine if it has the right interface; instead, the method or attribute is simply called or used (“If it looks like a duck and quacks like a duck, it must be a duck.”) By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution.*

— Python Glossary

Simply speaking, the program should allow you to swap data structures as long as the same syntax still makes sense. In C, for example, you have to define functions like the following:

```c
float fsquare(float x)
{
    return x * x;
};

int isquare(int x)
{
    return x * x;
};
```

While the operation `x * x` is identical for integers and floating-point numbers, a function taking an integer argument and a function taking a floating-point argument are not the same. Because types are static in C, we must define two functions although they perform the same logic. In Python, types are dynamic; hence we can define the corresponding function as:

```python
def square(x):
    return x * x
```

This feature indeed gives us tremendous power and convenience. For example, from scikit-learn, we have a function to do cross validation:

```
# evaluate a perceptron model on the dataset
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.linear_model import Perceptron
# define dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=10, n_redundant=0, r
# define model
model = Perceptron()
# define model evaluation method
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
# summarize result
print('Mean Accuracy: %.3f (%.3f)' % (mean(scores), std(scores)))
```

But in the above, the `model` is a variable of a scikit-learn-model object. It doesn't matter if it is a perceptron model as in the above, a decision tree, or a support vector machine model. What matters is that inside the `cross_val_score()` function, the data will be passed onto the model with its `fit()` function. Therefore, the model must implement the `fit()` member function, and the `fit()` function behaves identically. The consequence is that the `cross_val_score()` function is not expecting any particular model type as long as it looks like one. If we are using Keras to build a neural network model, we can make the Keras model look like a scikit-learn model with a wrapper:

```
# MLP for Pima Indians Dataset with 10-fold cross validation via sklearn
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score
from sklearn.datasets import load_diabetes
import numpy

# Function to create model, required for KerasClassifier
def create_model():
    # create model
    model = Sequential()
    model.add(Dense(12, input_dim=8, activation='relu'))
    model.add(Dense(8, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-i
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(build_fn=create_model, epochs=150, batch_size=10, verbose=0)
# evaluate using 10-fold cross validation
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=seed)
results = cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

In the above, we used the wrapper from Keras. Other wrappers exist, such as scikeras. All it does is to make sure the **interface** of the Keras model looks like a scikit-learn classifier so you can make use of the `cross_val_score()` function. If we replace the `model` above with:

```
model = create_model()
```

then the scikit-learn function will complain as it cannot find the `model.score()` function.

Similarly, because of duck typing, we can reuse a function that expects a list for a NumPy array or pandas series because they all support the same indexing and slicing operation. For example, we fit a time series with ARIMA as follows:

```
from statsmodels.tsa.statespace.sarimax import SARIMAX
import numpy as np
import pandas as pd

data = [266.0,145.9,183.1,119.3,180.3,168.5,231.8,224.5,192.8,122.9,336.5,185.9,
        194.3,149.5,210.1,273.3,191.4,287.0,226.0,303.6,289.9,421.6,264.5,342.3,
        339.7,440.4,315.9,439.3,401.3,437.4,575.5,407.6,682.0,475.3,581.3,646.9]
model = SARIMAX(y, order=(5,1,0))
res = model.fit(disp=False)
print("AIC = ", res.aic)

data = np.array(data)
model = SARIMAX(y, order=(5,1,0))
res = model.fit(disp=False)
print("AIC = ", res.aic)

data = pd.Series(data)
model = SARIMAX(y, order=(5,1,0))
res = model.fit(disp=False)
print("AIC = ", res.aic)
```

The above should produce the same AIC scores for each fitting.

Scopes and Name Space in Python

In most languages, variables are defined in a limited scope. For example, a variable defined inside a function is accessible only inside that function:

```
from math import sqrt

def quadratic(a,b,c):
    discrim = b*b - 4*a*c
    x = -b/(2*a)
    y = sqrt(discrim)/(2*a)
    return x-y, x+y
```

The **local variable** `discrim` is in no way accessible if we are not inside the function `quadratic()`. Moreover, this may be surprising for someone:

```
a = 1

def f(x):
    a = 2 * x
    return a

b = f(3)
print(a, b)
```

```
1 6
```

We defined the variable a outside function `f`, but inside `f`, variable a is assigned to be `2 * x`. However, the a inside the function and the one outside are unrelated except for the name. Therefore, as we exit from the function, the value of a is untouched. To make it modifiable inside function `f`, we need to declare the name a as `global` to make it clear that this name should be from the **global scope**, not the **local scope**:

```
a = 1

def f(x):
    global a
    a = 2 * x
    return a

b = f(3)
print(a, b)
```

```
6 6
```

However, we may further complicate the issue when introducing the **nested scope** in functions. Consider the following example:

```
a = 1

def f(x):
    a = x
    def g(x):
        return a * x
    return g(3)

b = f(2)
print(b)
```

```
6
```

The variable a inside function `f` is distinct from the global one. However, when inside g, since there is never anything written to a but merely read from it, Python will see the same a from the nearest scope, i.e., from function `f`. The variable x, however, is defined as an argument to the function g, and it takes the value 3 when we called `g(3)` instead of assuming the value of x from function `f`.

NOTE: If a variable has any value assigned to it **anywhere** in the function, it is defined in the local scope. And if that variable has its value read from it before the assignment, an error is raised rather than using the value from the variable of the same name from the outer or global scope.

This property has many uses. Many implementations of memoization decorators in Python make clever use of the function scopes. Another example is the following:

```python
import numpy as np

def datagen(X, y, batch_size, sampling_rate=0.7):
    """A generator to produce samples from input numpy arrays X and y
    """
    # Select rows from arrays X and y randomly
    indexing = np.random.random(len(X)) < sampling_rate
    Xsam, ysam = X[indexing], y[indexing]

    # Actual logic to generate batches
    def _gen(batch_size):
        while True:
            Xbatch, ybatch = [], []
            for _ in range(batch_size):
                i = np.random.randint(len(Xsam))
                Xbatch.append(Xsam[i])
                ybatch.append(ysam[i])
            yield np.array(Xbatch), np.array(ybatch)

    # Create and return a generator
    return _gen(batch_size)
```

This is a **generator function** that creates batches of samples from the input NumPy arrays `X` and `y`. Such a generator is acceptable by Keras models in their training. However, for reasons such as cross validation, we do not want to sample from the entire input arrays `X` and `y` but a **fixed** subset of rows from them. The way we do it is to randomly select a portion of rows at the beginning of the `datagen()` function and keep them in `Xsam`, `ysam`. Then in the inner function `_gen()`, rows are sampled from `Xsam` and `ysam` until a batch is created. While the lists `Xbatch` and `ybatch` are defined and created inside the function `_gen()`, the arrays `Xsam` and `ysam` are not local to `_gen()`. What's more interesting is when the generator is created:

```python
X = np.random.random((100,3))
y = np.random.random(100)

gen1 = datagen(X, y, 3)
gen2 = datagen(X, y, 4)
print(next(gen1))
print(next(gen2))
```

```
(array([[0.89702235, 0.97516228, 0.08893787],
       [0.26395301, 0.37674529, 0.1439478 ],
       [0.24859104, 0.17448628, 0.41182877]]), array([0.2821138 , 0.87590954, 0.96646776]))
(array([[0.62199772, 0.01442743, 0.4897467 ],
       [0.41129379, 0.24600387, 0.53640666],
       [0.02417213, 0.27637708, 0.65571031],
       [0.15107433, 0.11331674, 0.67000849]]), array([0.91559533, 0.84886957, 0.30451455, 0.
```

The function `datagen()` is called two times, and therefore two different sets of `Xsam`, `yam` are created. But since the inner function `_gen()` depends on them, these two sets of `Xsam`, `ysam` are in memory concurrently. Technically, we say that when `datagen()` is called, a **closure** is created with the specific `Xsam`, `ysam` defined within, and the call to `_gen()` is accessing that closure. In other words, the scopes of the two incarnations of `datagen()` calls coexist.

In summary, whenever a line of code references a name (whether it is a variable, a function, or a module), the name is resolved in the order of the LEGB rule:

1. Local scope first, i.e., those names that were defined in the same function
2. Enclosure or the "nonlocal" scope. That's the upper-level function if we are inside the nested function.
3. Global scope, i.e., those that were defined in the top level of the same script (but not across different program files)
4. Built-in scope, i.e., those created by Python automatically, such as the variable `__name__` or functions `list()`

Want to Get Started With Python for Machine Learning?

Take my free 7-day email crash course now (with sample code).

Click to sign-up and also get a free PDF Ebook version of the course.

Download Your FREE Mini-Course

Investigating the type and scope

Because the types are not static in Python, sometimes we would like to know what we are dealing with, but it is not trivial to tell from the code. One way to tell is using the `type()` or `isinstance()` functions. For example:

```python
import numpy as np

X = np.random.random((100,3))
print(type(X))
print(isinstance(X, np.ndarray))
```

```
<class 'numpy.ndarray'>
True
```

The `type()` function returns a type object. The `isinstance()` function returns a Boolean that allows us to check if something matches a particular type. These are useful in case we need to know what type a variable is. This is useful if we are debugging a code. For example, if we pass on a pandas dataframe to the `datagen()` function that we defined above:

```python
import pandas as pd
import numpy as np

def datagen(X, y, batch_size, sampling_rate=0.7):
    """A generator to produce samples from input numpy arrays X and y
    """
    # Select rows from arrays X and y randomly
    indexing = np.random.random(len(X)) < sampling_rate
    Xsam, ysam = X[indexing], y[indexing]

    # Actual logic to generate batches
    def _gen(batch_size):
        while True:
            Xbatch, ybatch = [], []
            for _ in range(batch_size):
                i = np.random.randint(len(Xsam))
                Xbatch.append(Xsam[i])
                ybatch.append(ysam[i])
            yield np.array(Xbatch), np.array(ybatch)

    # Create and return a generator
    return _gen(batch_size)

X = pd.DataFrame(np.random.random((100,3)))
y = pd.DataFrame(np.random.random(100))

gen3 = datagen(X, y, 3)
print(next(gen3))
```

Running the above code under the Python's debugger `pdb` will give the following:

```
> /Users/MLM/ducktype.py(1)<module>()
-> import pandas as pd
(Pdb) c
Traceback (most recent call last):
  File "/usr/local/lib/python3.9/site-packages/pandas/core/indexes/range.py", line 385, in g
    return self._range.index(new_key)
ValueError: 1 is not in range

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "/usr/local/Cellar/python@3.9/3.9.9/Frameworks/Python.framework/Versions/3.9/lib/pyth
    pdb._runscript(mainpyfile)
  File "/usr/local/Cellar/python@3.9/3.9.9/Frameworks/Python.framework/Versions/3.9/lib/pyth
    self.run(statement)
  File "/usr/local/Cellar/python@3.9/3.9.9/Frameworks/Python.framework/Versions/3.9/lib/pyth
    exec(cmd, globals, locals)
  File "<string>", line 1, in <module>
  File "/Users/MLM/ducktype.py", line 1, in <module>
    import pandas as pd
  File "/Users/MLM/ducktype.py", line 18, in _gen
    ybatch.append(ysam[i])
  File "/usr/local/lib/python3.9/site-packages/pandas/core/frame.py", line 3458, in __getite
    indexer = self.columns.get_loc(key)
  File "/usr/local/lib/python3.9/site-packages/pandas/core/indexes/range.py", line 387, in g
    raise KeyError(key) from err
KeyError: 1
Uncaught exception. Entering post mortem debugging
Running 'cont' or 'step' will restart the program
> /usr/local/lib/python3.9/site-packages/pandas/core/indexes/range.py(387)get_loc()
-> raise KeyError(key) from err
(Pdb)
```

We see from the traceback that something is wrong because we cannot get `ysam[i]`. We can use the following to verify that `ysam` is indeed a Pandas DataFrame instead of a NumPy array:

```
(Pdb) up
> /usr/local/lib/python3.9/site-packages/pandas/core/frame.py(3458)__getitem__()
-> indexer = self.columns.get_loc(key)
(Pdb) up
> /Users/MLM/ducktype.py(18)_gen()
-> ybatch.append(ysam[i])
(Pdb) type(ysam)
<class 'pandas.core.frame.DataFrame'>
```

Therefore we cannot use `ysam[i]` to select row `i` from `ysam`. What can we do in the debugger to verify how we should modify our code? There are several useful functions you can use to investigate the variables and the scope:

- `dir()` to see the names defined in the scope or the attributes defined in an object
- `locals()` and `globals()` to see the names and values defined locally and globally, respectively.

For example, we can use `dir(ysam)` to see what attributes or functions are defined inside `ysam`:

```
(Pdb) dir(ysam)
['T', '_AXIS_LEN', '_AXIS_ORDERS', '_AXIS_REVERSED', '_AXIS_TO_AXIS_NUMBER',
...
'iat', 'idxmax', 'idxmin', 'iloc', 'index', 'infer_objects', 'info', 'insert',
'interpolate', 'isin', 'isna', 'isnull', 'items', 'iteritems', 'iterrows',
'itertuples', 'join', 'keys', 'kurt', 'kurtosis', 'last', 'last_valid_index',
...
'transform', 'transpose', 'truediv', 'truncate', 'tz_convert', 'tz_localize',
'unstack', 'update', 'value_counts', 'values', 'var', 'where', 'xs']
(Pdb)
```

Some of these are attributes, such as `shape`, and some of these are functions, such as `describe()`. You can read the attribute or invoke the function in `pdb`. By carefully reading this output, we recalled that the way to read row `i` from a DataFrame is through `iloc`, and hence we can verify the syntax with:

```
(Pdb) ysam.iloc[i]
0    0.83794
Name: 2, dtype: float64
(Pdb)
```

If we call `dir()` without any argument, it gives you all the names defined in the current scope, e.g.,

```
(Pdb) dir()
['Xbatch', 'Xsam', '_', 'batch_size', 'i', 'ybatch', 'ysam']
(Pdb) up
> /Users/MLM/ducktype.py(1)<module>()
-> import pandas as pd
(Pdb) dir()
['X', '__builtins__', '__file__', '__name__', 'datagen', 'gen3', 'np', 'pd', 'y']
(Pdb)
```

where the scope changes as you move around the call stack. Similar to `dir()` without argument, we can call `locals()` to show all locally defined variables, e.g.,

```
(Pdb) locals()
{'batch_size': 3, 'Xbatch': ...,
 'ybatch': ..., '_': 0, 'i': 1, 'Xsam': ...,
 'ysam': ...}
(Pdb)
```

Indeed, `locals()` returns you a `dict` that allows you to see all the names and values. Therefore, if we need to read the variable `Xbatch`, we can get the same with `locals()["Xbatch"]`. Similarly, we can use `globals()` to get a dictionary of names defined in the global scope.

This technique is beneficial sometimes. For example, we can check if a Keras model is “compiled” or not by using `dir(model)`. In Keras, compiling a model is to set up the loss function for training and build the flow for forward and backward propagations. Therefore, a compiled model will have an extra attribute `loss` defined:

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([
    Dense(5, input_shape=(3,)),
    Dense(1)
])

has_loss = "loss" in dir(model)
print("Before compile, loss function defined:", has_loss)

model.compile()
has_loss = "loss" in dir(model)
print("After compile, loss function defined:", has_loss)
```

```
Before compile, loss function defined: False
After compile, loss function defined: True
```

This allows us to put an extra guard on our code before we run into an error.