Python - Missing Bits

Barry Rowlingson

December 1, 2021

This document lists a few bits of Python syntax that I've not covered in the lectures. Some are things you probably haven't seen and might not ever use. Others are useful tools to have in your programming toolbox.

1 PEP 8

The Python language develops through a process of PEPs - Python Enhancement Proposals. PEP number 8 discusses the style that Python code should be written in - how many spaces to indent blocks, when to use upper case, lower case, or mixed case, how to break long lines and so on. Code written to conform to PEP 8 will nearly always look good, and be more comprehensible to other Python coders.

There are ways to test your code for PEP 8 style conventions, and even to reformat code to comply. You might find these in your code editor or development environment, or you may need to install them separately. But if your code looks vaguely like commonly found Python code, you are probably getting close to it. I can be fairly loose with PEP 8 compliance myself...

2 Tuples and Lists

Python has two main base classes for storing ordered sequences - the tuple and the list. The main difference is that lists are *mutable* but tuples are not. This means you can change the values in a list but not in a tuple. Lists are constructed with square brackets and tuples with round ones, and they are printed with those brackets too.

```python
a_list = [1,2,3]
a_tuple = (1,2,3)
print(f"{a_list=}")
print(f"{a_tuple=}")
a_list[1] = 22 # can change a list element
print(f"{a_list=}")
a_tuple[1] = 99 # cant change a tuple element
```

```
a_list=[1, 2, 3]
a_tuple=(1, 2, 3)
a_list=[1, 22, 3]
```

```
---------------------------------------------------------------------------TypeError
Traceback (most recent call last)/tmp/ipykernel_1237524/2496353001.py
in <module>
      5 a_list[1] = 22 # can change a list element
      6 print(f"{a_list=}")
----> 7 a_tuple[1] = 99 # cant change a tuple element
TypeError: 'tuple' object does not support item assignment
```

A tuple of length one would seem to be created with `atuple = (99)` but that results in a scalar numeric value. Instead you use a trailing comma: `atuple = (99,)`.

3 Unpacking

If a function returns a tuple or a list, it can be returned split into separate objects. Here's an example:

```python
def compound():
  return [1, (2,3,4), "This"]

a, v, status = compound() # unpack result into three objects

print(f"{a = } ; {v = } ; {status = }")
```

```
a = 1 ; v = (2, 3, 4) ; status = 'This'
```

One neat trick is to use this to swap the values of two objects:

```python
a = 1
b = [2, 3, 4, 5]

a,b = b,a

print(f"{a = } ; {b = }")
```

```
a = [2, 3, 4, 5] ; b = 1
```

You can also use this to combine short related calls onto one line:

```python
import math
r, theta = 2, math.pi/5
x, y = r*math.sin(theta), r*math.cos(theta)
print(f"{r=},{theta=:.2f} -> {x=:.2f},{y=:.2f}")
```

```
r=2,theta=0.63 -> x=1.18,y=1.62
```

4 The Self-Modifying Operators

Taking a cue from the C and C++ languages, Python has functions that save writing the same things twice when doing something like `a = a + 1`.

```python
a = 99
a += 1
print(f"{a = }")
```

```
a = 100
```

5 Lambda Expressions

Python lets you create small functions in inline code without having to define them with a `def` section. You can use them wherever you'd put the name of a function. For example, suppose we want to get the even numbers from a list, we might write a function and use `filter`:

```python
a_list = [4,3,6,5,2,8,7]
def even(x):
  return x % 2 == 0
e_list = list(filter(even, a_list))
print(f"Evens are {e_list}")
```

```
Evens are [4, 6, 2, 8]
```

We can save code by using a *lambda* expression:

```
e_list = list(filter(lambda x: x % 2 == 0, a_list))
print(f"Evens are {e_list}")
```

```
Evens are [4, 6, 2, 8]
```

The concision here comes from not needing parentheses or a `return` statement, but a lambda expression should only be used in simple cases. If the logic gets too complex, put it in a function and give it a name!

6 Filtering in a List Comprehension

A list comprehension is a mapping over the values in a list:

```
a_list = [1,3,7,5,6,9,23,8]
b_list = [z * 2 for z in a_list]
print(f"{b_list = }")
```

```
b_list = [2, 6, 14, 10, 12, 18, 46, 16]
```

A list comprehension can have an extra "if" part that filters values before operating on them:

```
b_list = [z * 2 for z in a_list if z > 5]
print(f"{b_list = }")
```

```
b_list = [14, 12, 18, 46, 16]
```

You can do a filter without changing the values this way:

```
b_list = [z for z in a_list if z > 5]
print(f"{b_list = }")
```

```
b_list = [7, 6, 9, 23, 8]
```

Note that these operations can also be done with `map` and `filter` functions:

```
# get all even numbers and divide by two:
b_list = map( lambda x: x/2, filter(lambda x: x%2==0, a_list))
print(f"{b_list = }")
```

```
b_list = <map object at 0x7f609b841b50>
```

These functions return an "iterable" object rather than a list of values. If you need to see the values, convert to a `list`:

```
b_list = list(b_list)
print(f"{b_list = }")
```

```
b_list = [3.0, 4.0]
```

7 Decorators

A decorator takes a function and modifies its behaviour. This makes it easy to add functionality to functions. For example, if you want to know when a function is being called you could edit the function and add a `print` call:

```
def squarey(x):
  print("Running: squarey")
  return x**2 + x
squarey(10)
```

```
Running: squarey
```

```
110
```

but this messes up your nice function, and if you wanted to do this for several functions it would require a lot of editing. Instead you can write a decorator, which is a special function that returns a function.

```
def shout(func):
    def wrapper(*args, **kwargs):
        print(f'Running: {func.__name__}')
        return func(*args, **kwargs)
    return wrapper
```

To use this, there's a special `@decorator` syntax:

```
@shout
def squarey(x):
  return x**2 + x
squarey(10)
```

```
Running: squarey
```

```
110
```

Python comes with a few decorators but most are supplied in add-on modules. They can do things like:

1. Check types of arguments
2. Log times of calls to functions
3. Time how long functions take to run
4. Check return values from functions

8 The `with` Statement

A common situation in computing is when you have to set something up, then do something, then reset the thing you first set up - for example to read from a file you first open the connection to the file, then you read from the connection, then you close the connection:

```
f = open("misc.ptexw")
first_line = f.readlines(1)[0]
print(first_line)
f.close()
```

```
% this is the first line
```

But what if something goes wrong in the code between the open and the close, and the close doesn't get executed? How can we guarantee that the connection is closed? One method is to use `try:` with a `finally:` clause:

```
try:
   f = open("misc.ptexw")
   print(f.readlines(1)[0])
finally:
   f.close()
```

```
% this is the first line
```

To support this in a neater way, Python now has the `with` block. The same code above can now be written in two lines:

```
with open("misc.ptexw") as f:
   print(f.readlines(1)[0])
```

```
% this is the first line
```

Here the `open` call is operating as a *context manager*, and knows to call the `close` method at the end of the block, hiding something compulsory that the user really doesn't want to think about doing. Other context managers exist and you can write your own. If you see a `with` block, then there's a context manager doing something for you there.

9 More Argument Specifications

9.1 "Starred" Arguments

Functions do not have to have a fixed number of arguments passed to them if they are defined with special `*args` or `**kwargs` arguments (these names are conventional, but the number of stars is important).

When a function is defined with these, any unnamed parameters not matched by explicit unnamed parameters are put in `args` as a tuple, and any keyword parameters not otherwise matched are put in `kwargs` as a dictionary.

```
def show_args(x, y, *args, **kwargs):
  print(f"{x = }, {y = }")
  print(f"{args = }")
  print(f"{kwargs = }")

show_args(1, 2, 3, 4, z="five", aa=6)
```

```
x = 1, y = 2
args = (3, 4)
kwargs = {'z': 'five', 'aa': 6}
```

9.2 "Slash" Arguments

You might see functions defined like this:

```
def slashfun(a, b, /, c):
   print(f"{a = } ; {b = } ; {c = }")
```

The forward slash mark in the definition marks a boundary between positional and keyword arguments. Those before the slash cannot be named in a call to the function:

```
slashfun(1, 2, 3) # this is fine
slashfun(a=1, b=2, c=3) # nah
```

```
a = 1 ; b = 2 ; c = 3
```

```
---------------------------------------------------------------------------TypeError
Traceback (most recent call last)/tmp/ipykernel_1237524/923953127.py
in <module>
      1 slashfun(1, 2, 3) # this is fine
----> 2 slashfun(a=1, b=2, c=3) # nah
TypeError: slashfun() got some positional-only arguments passed as
keyword arguments: 'a, b'
```

This is a fairly new addition to Python and I only knew about it when I saw it in a help text, for example in `math.sin`:

```
sin(x, /)
    Return the sine of x (measured in radians).
```

This means you can't call it as like `sin(x=6.28)`, but as `sin(6.28)` instead.

10 Generators

A generator is a function that yields the next value in a sequence. Any function that uses `yield` instead of `return` can be used to make a generator. Think of it like a function that returns a value when it gets to a `yield` but resumes from that point next time its called.

```
def g1():
   yield 123
   yield 321
   yield 132
```

Because this needs to save its state, you can't use it quite directly. Instead, this is a generator function or generator `factory`, and calling it returns a generator, or an instance of the generator, which you can use.

Here's the result of calling `next` on an instance of the above generator:

```
gg = g1()
print(next(gg))
print(next(gg))
print(next(gg))
```

```
123
321
132
```

This can be very useful in certain circumstances. Firstly, its possible to have a generator that produces an infinite series, which would be impractical to have in a list. Then any code that wanted part of that series would get just those values computed:

```
def powers():
    i = 1
    while True:
       yield i
       i = i * 2

pg = powers()
power10 = [next(pg) for i in range(10)]
print(f"{power10=}")
```

```
power10=[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

If you have a generator that creates a finite sequence you can iterate over it using Python's usual iterator systems:

```
def powersN(N):
  p = 1
  for i in range(N):
    yield p
    p *= 2

p5 = powersN(5)
for p in p5:
  print(f"{p=} ;", end='')
```

```
p=1 ;p=2 ;p=4 ;p=8 ;p=16 ;
```

Note now the generator `p5` has completed, and so if you try the loop again it will not run:

```
print("iterating over p5...")
for p in p5:
  print(f"{p=} ;", end='')
print("done iterating over p5. Did anything happen?")
```

```
iterating over p5...
done iterating over p5. Did anything happen?
```

To rerun the generator you get a fresh instance of the generator from the generator function:

```
p5 = powersN(5)
for p in p5:
  print(f"{p=} ;", end='')
```

```
p=1 ;p=2 ;p=4 ;p=8 ;p=16 ;
```

Alternatively you can use `list` on a generator to get all its values, but this is often not needed if you are going to loop over the values.

```
p3 = powersN(3)
all3 = list(p3)
print(f"{all3 = }")
```

```
all3 = [1, 2, 4]
```

11 Type Hint Annotations

Type hint annotations are another modern addition to Python that you might see and at first look technical and confusing. Here's an example:

```
def solve(x: float, n: int) -> float:
   return x*n
```

What's happening here is that the programmer has specified "hints" as to what type of value each of the arguments should be, and what type the returned value should be. But (currently) Python doesn't check or enforce any of this. I can still pass a character string as `x` and it works.

```
s = solve( 3, 2) # multiplies numbers
print(s)
s = solve("This", 3) # repeats strings
print(s)
```

```
6
ThisThisThis
```

12 The Walrus Operator

Assignments in Python, like `x = 123`, are statements that don't return a value. You can't use them anywhere except the start of a line. The walrus operator allows you to make assignments that do return a value, and this can be a useful aid to making code more readable.

The operator, with a stretch of the imagination, looks like a walrus: `:=` as the dots look like eyes and the equals sign is supposed to look like its tusks. Here's a sample usage:

```
x = 100
a = 1 # this is getting overwritten by the walrus
z = (a:=x) > 50
print(f"{z = } and {a = }")
```

```
z = True and a = 100
```

Notice the line with the walrus operator has done two things: it has set the value of `a` to the value of `x` and it has set `z` to whether `a` is then greater than 50. The value of a walrus operator is the value assigned to the object on the left of it. The above code is the same as:

```
x = 100
a = 1
a = x
z = a > 50
print(f"{z = } and {a = }")
```

```
z = True and a = 100
```

Here's another example where we get a random number and test if it is over 0.5 in one line, keeping the random number in `z` for later use:

```
from random import uniform
if (z:=uniform(0,1)) > 0.5:
   print(f"Large {z=}")
else:
   print(f"Small {z=}")
```

```
Large z=0.7187508978004764
```

You might find the walrus operator in other contexts too, so watch out.