More Python

Principles of Programming Languages

Colorado School of Mines

https://lambda.mines.edu

LGA & Review

Learning Group Activity

Review the learning group activity with your group members:

1. What questions, comments, or snarky remarks did you have on the official Python style guide (PEP 8)?
2. Demonstrate what you made in Python.

Review: Slicing

```
mylist = [1, 2, 3, 4]

# syntax is [start:stop:step], step optional
mylist[1:3]     # => [2, 3]

# unused parameters can be ommited
mylist[::-1]    # => [4, 3, 2, 1]

# without the first element
mylist[1:]      # => [2, 3, 4]

# without the last element
mylist[:-1]     # => [1, 2, 3]
```

Review: Tuple Expansion & Collection

Multiple assignments work like so:

```
names = ("R. Stallman", "L. Torvalds", "B. Joy")
a, b, c = names
```

* can be used to collect a tuple:

```
# drop the lowest and highest grade
grades = (79, 81, 93, 95, 99)
lowest, *grades, highest = grades
```

The same can be done to expand a tuple in a function call:

```
# Each grade becomes a separate argument
print(*grades)
```

Functions

Review: Functions

To define a function in Python, use the def syntax:

```
def myfun(arg1, arg2, arg3):
    if arg1 == 'hello':
        return arg2
    return arg3
```

Even if your function does not take arguments, you still need the parentheses:

```
def noargs():
    print("I'm all lonely without arguments...")
```

Keyword Arguments

When we define a function in Python we may define **keyword arguments**. Keyword arguments differ from *positional arguments* in that keyword arguments:

- Take a default value if unspecified
- Can be placed either in order or out of order:
  - **In order:** arguments are assigned in the order of the function definition
  - **Out of order:** the argument name is written in the call
- Positional and keyword arguments can be mixed, so as long as the positional arguments go first.

Keyword Arguments: Example

```
def point_twister(x, y=1, z=0):
    return x + 2*z - y

# all of these are valid calls
print(point_twister(1, 2, 3))        # x=1, y=2, z=3
print(point_twister(1, 2))           # x=1, y=2, z=0
print(point_twister(1))              # x=1, y=1, z=0
print(point_twister(1, z=2, y=0))    # x=1, y=0, z=2
print(point_twister(1, z=2))         # x=1, y=1, z=2
```

Style Note

PEP 8 says that we should place spaces around our "=" in assignments, but these are not assignments, and should be written without spaces around the "=".

Passing a Dictionary as the Keyword Arguments

Just like a tuple or list can be expanded to the positional arguments of a function call using `*some_tuple`, a dictionary can be expanded to the keyword arguments of a function using `**some_dict`. For example:

```
my_point = {'x': 10, 'y': 15, 'z': 20}
print(point_twister(**my_point))
```

*args and **kwargs

Python allows you to define functions that take a variable number of positional (*args) or keyword (**kwargs) arguments. In principle, this really just works like tuple expansion/collection.

```
def crazyprinter(*args, **kwargs):
    for arg in args:
        print(arg)
    for k, v in kwargs.items():
        print("{}={}".format(k, v))

crazyprinter("hello", "cheese", bar="foo")
# hello
# cheese
# bar=foo
```

The names args and kwargs are merely a convention. For example, you could use the names rest and kwds instead if you wanted.

*args and **kwargs: Another Example

```
def fancy_args(a, b, *args, c=10, **kwargs):
    print("a is", a)
    print("b is", b)
    print("c is", c)
    print("args is", args)
    print("kwargs is", kwargs)

fancy_args(1, 2, 3, 4, c=15, d=16, e=17)
# a is 1
# b is 2
# c is 15
# args is (3, 4)
# kwargs is {'d': 16, 'e': 17}
```

Anonymous Functions

The Python keyword `lambda` creates an anonymous function. The syntax is:

```
lambda arg1, arg2, ...: result
```

For example:

```
double = lambda x: x * 2
```

But is it really cleaner?

A lot of `lambda` functions can make your code hard to read. But there does exist the occasion a `lambda` will make your life easier (`defaultdict` example).

Generators

Generator Functions

A special kind of function exists called a **generator function**. A generator function yields values rather than returning them: rather than exiting the function call, the function continues to run and yield more values.

```
def one_to(stop):
    x = 1
    while x <= stop:
        yield x
        x += 1
```

Using Generator Functions

Calling a generator function produces a **generator object**:

```
my_gen = one_to(5)
```

Calling `next` on the generator object gets us the next thing it yields:

```
print(next(my_gen))      # => 1
print(next(my_gen))      # => 2
print(next(my_gen))      # => 3
print(next(my_gen))      # => 4
```

When the function exits, calling `next` raises a `StopIteration` exception:

```
print(next(my_gen))      # => 5
print(next(my_gen))      # raises StopIteration
```

But we rarely use next directly...

`for` loops can use it for us:

```
# Prints 1, 2, then 3
# The loop exits on StopIteration
for x in one_to(3):
    print(x)
```

We can create lists, sets, and many other things from generator objects:

```
list(one_to(8))   # => [1, 2, 3, 4, 5, 6, 7, 8]
set(one_to(8))    # => {1, 2, 3, 4, 5, 6, 7, 8}
tuple(one_to(8))  # => (1, 2, 3, 4, 5, 6, 7, 8)
```

Generator Functions: Another Example

We could define a function (similar to) range that we talked about last time:

```
def range(start, stop, step=1):
    i = start
    while i < stop:
        yield i
        i += step
```

Generator Expressions (Anonymous Generator Functions)

A generator function can be created anonymously:

```
(x * 2 for x in nums if x % 2 == 0)
```

Consider this similar to the following Haskell list comprehension:

```
[x * 2 | x <- nums, x `mod` 2 == 0]
```

There's three parts to a generator expression:

1. The output expression which computes each value, this is `x * 2` above
2. Preforming something for every element in a sequence, this is `for x in nums` above
3. Selecting a subset of elements to operate on, this is `if x % 2 == 0` above

GEs: Multiple Loops

Multiple loops can be written inside of a GE, and the loops will be evaluated *outside-in*:

```
>>> gen = ((x, y) for x in range(15)
                  if happy(x)
                  for y in range(2))
>>> list(gen)
[(1, 0), (1, 1),
 (7, 0), (7, 1),
 (10, 0), (10, 1),
 (13, 0), (13, 1)]
```

Note

The function happy is not included in Python, but can be found on the course website.

GEs: Syntax Details

If a GE is the only argument to a function call, the second set of parentheses can be omitted:

```
print("The smallest was:",
    min(input("Give me a number: ") for _ in range(5)))
```

You could use this to build lists or sets, for example:

```
list(x + 1 for x in range(3))   # => [1, 2, 3]
set(x + 1 for x in range(3))    # => {1, 2, 3}
```

But Python provides a more convenient syntax for that...

Comprehensions

A **list comprehension** is written as a GE with brackets. Think of it as a eager generator expression:

```
[x * 2 for x in nums if x % 2 == 0]
```

Similarly, a **set comprehension** is written as a GE with braces:

```
{x % 7 for x in range(0, 20, 5)}
```

And we can even write **dictionary comprehensions**:

```
{x: f(x) for x in range(10)}
```

Applications of GEs

- File readers

```
reader = (float(line) for line in f)
while event_queue:
    process_event(next(reader))
```

- Hash function pRNGs

```
rng = (h(x)/MAX_HASH for x in count())
```

- **The possibilities are endless!** I use GEs and comprehensions all the time since they are highly expressive.

Modules

Modules

Often times, we wish to break our software into several files and namespaces. Python provides a very simple way to do this:

1. Write your functions in a file called `somemodule.py`
2. Type `import somemodule` at the top of your program.
3. You'll now have access to an object named `somemodule` whose members are the objects from `somemodule.py`

See `happy.py` on the course website for a simple example.

Bringing Things Into our Namespace

Typing `import somemodule` will provide you with a module object which you can access members, but does not declare any new variables in your namespace except for the `somemodule` object.

To bring in certain members, you can use a `from` statement:

```
from somemodule import f1, f2
```

Aliasing

Often times we don't want to call the module in our namespace what the filename is, so we can use as to rename:

```
import somemodule as mod

mod.f1(...)
```

Or, using a `from`:

```
from somemodule import f1 as somefunc

somefunc(...)
```

More Complex Modules

We may wish to make very complex modules, which are composed of multiple files. To do so:

1. Create a directory with the desired module name (e.g., `somemodule`)
2. Put a file in that directory named `__init__.py`. When `import somemodule` is typed, this is the file that will be imported.
3. Create other parts of the module under other file names, these can be imported by typing `import somemodule.somefile`. From within our module, we can type `from .somefile import x`.

Functional Programming

Partial Application

The `partial` function from the `functools` library provides us with a partial applicator:

```
from functools import partial

value_print = partial(print, "The value is: ")

value_print(10)
# The value is: 10
```

min/max

The `min` and `max` functions select the minimum/maximum element from an sequence or generator, optionally based on a `key` function:

```
closest_point = min(points, key=partial(dist, ref))
```

Compare to the equivalent procedural code:

```
closest_dist = float('inf')
closest_point = None
for p in points:
    d = dist(ref, p)
    if d < closest_dist:
        closest_dist = d
        closest_point = p
```

You tell me which code snippet is more expressive. ;)

zip

Python takes a little inspiration from Haskell and provides a `zip` generator function which yields pairwise tuples from each of its arguments.

```
small = [1, 2, 3]
med = [10, 20, 30]
large = [100, 200, 300]
for a, b, c in zip(small, med, large):
    print(a, b, c)
# 1 10 100
# 2 20 200
# 3 30 300
```

Pro Tip: Iterating over the columns of a row-major 2D list

```
for col in zip(*arr)
```

More Haskell Inspiration

- `map(f, *sequences)` is a generator function that applies a function to a sequence of elements. Anything that can be done with `map` could also be done using a GE (and potentially `zip`), so your choice on whether to use this.
- `reduce(f, seq)` is the general-case reduction function: it takes a function and folds it across the sequence. (from `functools`)

Thursday

There will be a quiz on the **Lambda Calculus** on Thursday. Make sure you take a stab at the practice problems on the course website.