More Python

Principles of Programming Languages

Colorado School of Mines

https://lambda.mines.edu
Generators
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.

```python
def one_to(stop):
    x = 1
    while x <= stop:
        yield x
        x += 1
```
Using Generator Functions

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

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

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

```python
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:

```python
print(next(my_gen))  # => 5
print(next(my_gen))  # raises StopIteration
```
But we rarely use `next` directly...

for loops can use it for us:

```python
# 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:

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

```python
def range(start, stop, step=1):
    i = start
    while i < stop:
        yield i
        i += step
```
A generator function can be created anonymously:

\[(x \times 2 \text{ for } x \text{ in } \text{nums} \text{ if } x \mod 2 == 0)\]

Consider this similar to the following set builder notation (from mathematics):

\[\{x \times 2 : x \in \text{nums} | x \mod 2 = 0\}\]

There’s three parts to a generator expression:

1. The output expression which computes each value, this is \(x \times 2\) above
2. Performing something for every element in a sequence, this is \(\text{for } x \text{ in } \text{nums}\) above
3. Selecting a subset of elements to operate on, this is \(\text{if } x \mod 2 == 0\) above
Multiple loops can be written inside of a GE, and the loops will be evaluated outside-in:

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

```python
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:

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

\[ [x \times 2 \text{ for } x \text{ in } \text{nums} \text{ if } x \% 2 == 0] \]

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

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

And we can even write dictionary comprehensions:

\{x: f(x) \text{ for } x \text{ in } \text{range}(10)\}
Applications of GEs

- File readers
  ```python
  reader = (float(line) for line in f)
  while event_queue:
    process_event(next(reader))
  ```

- Hash function pRNGs
  ```python
  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
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.
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:

```python
from somemodule import f1, f2
```
Often times we don’t want to call the module in our namespace what the filename is, so we can use as to rename:

```python
import somemodule as mod
mod.f1(...)
```

Or, using a from:

```python
from somemodule import f1 as somefunc
somefunc(...)```

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.