# More Python

**Principles of Programming Languages**

Colorado School of Mines

https://lambda.mines.edu

# Generators

# Generator Functions

A special kind of function exists called a **generator function** (also referred to as a *coroutine*). 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
```

# Generators Are Lazy

Generator functions do not compute the next value until asked! Take the following example:

```
def fib_seq():
    a, b = 0, 1
    yield a
    yield b

    while True:
        yield a + b
        a, b = b, a + b

fib = fib_seq()
print(next(fib), next(fib), next(fib), next(fib), next(fib))  # 0 1 1 2 3
```

This property is called *lazy evaluation*.

# 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 set builder notation (from mathematics):

$$\{x \times 2 : x \in \texttt{nums} \mid x \bmod 2 = 0\}$$

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 y in range(7)
                  if y % 3 == 0
                  for x in range(y))

>>> list(gen)
[(0, 3), (1, 3), (2, 3),
 (0, 6), (1, 6), (2, 6), (3, 6), (4, 6), (5, 6)]
```

**Note**

The syntax for dependent `for` statements is a bit strange. Keep in mind that the variable must exist before it is used.

# 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`

# 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(...)
```

**Note**

You will often see aliasing used with `itertools`:

```
import itertools as it
```

# 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`.

# Self Learning

I think the best way to become familiar with Python (or any language) is to use it! Here are some ideas which can help you:

- **Kattis:** `https://open.kattis.com/`
- Implement a data structures/algorithms assignment in Python.
- If you learn better by reading: `https://pythontips.com/2013/09/01/best-python-resources/` has some good resources.