CSE 20 12-2-21

- Final exam: Mon Dec. 6
- SETs
- grading
  - lab 4, lab 5, Pas → fri. PM
  - Pa6 → sun. PM
  - Pa7 → fri. of finals wk.

list (and other) comprehensions

L = [exp for x in iterable <u>if cond</u>]
optional

Ex. [x**2 for x in range(1, 4)]

→ [1, 4, 9]

Ex. `[x**2 for x in range(1,11) if x%3==1]`

→ `[1, 16, 49, 100]`

Same as

```
L = []
for x in range(1, 11):
    if x % 3 == 1
        L.append(x**2)
    #end
#end

# L = [1, 16, 49, 100]
```

Generator expressions & Generator objects

first: iterable objects

anything you can put in a for loop

```
for x in iterable:
    #do something
```

iterable: list, tuple, string, range, enumerate, file

Defn

A class is iterable iff it implements the methods

- `__iter__()` } the iterable protocal
- `__next__()` }

These implementations define behavior of built-in fcns: iter(), next()

Any iterable object can be used to create an iterator object

Ex. `L = [1, 2, 3]  # iterable obj`

`I = iter(L)  # iterator obj.`

An iterator obj is itself iterable

```
for x in I:
    print(x)
```

we can also iterate over I using next()

```
next(I)  # returns 1
next(I)  # returns 2
next(I)  # .. 3
next(I)  # raises StopIteration exception.
```

or

sentinal value

```
next(I, 'foo') # 1
next(I, 'foo') # 2
next(I, 'foo') # 3
next(I, 'foo') # returns 'foo'
```

we cannot call next() on an iterable obj that is not an iterator obj.

```
nex(L) # raises TypeError
```

An iterator can only be used once.

```
I = iter(L) # L=[1, 2, 3]
for x in I:
    print(x)
next(I) # stopIteration exception.
```

A for loop is actually using an iterator:

```
for x in L:
    print(x)
#end
```

is equivalent to

```
I = iter(L)
x = next(I, 'foo')
while x != 'foo':
    print(x)
    x = next(I, 'foo')
#end
```

or any other sentinal value

The in operator can be used on an iterator, consuming some or all elements

```
I = iter(L)  # L = [1, 2, 3]

1 in I  # True
next(I)  # 2
1 in I  # False, past 1
next(I)  # StopIteration
```

Can also turn an iterator into an iterable:

```
I = iter(L)  # L = [1, 2, 3]
T = tuple(I)
print(T)  # (1, 2, 3)
next(I)  # StopIteration, now consumed
```

You only use what's left in I

```
I = iter(L)
next(I) # 1
S = set(I) # {2, 3}
```

or

```
I = iter(L)
4 in I # false
S = set(I) # {}
```

A Generator is a type of iterator that can be taylored to a specific need.

Generator expressions:

Ex. `G = ( x**2 for x in range(1, 4) )`

```
next(G) # 1
next(G) # 4
next(G) # 9
# now exhausted
```

when we use a generator obj. in a constructor (list, tuple, set...) its called Comprehension.

```
S = { x**2 for x in range(5) }
# S = { 0, 1, 4, 9, 16 }
```

or

```
S = set( x**2 for x in range(5) )
# same thing
```

Can use logic to create more complicated Generator objects.

```
G = (x**2 for x in range(8) if x%2==1)
```

optional.

```
# Produces: 1, 9, 25, 49
```

if G is used in a comprehension it is consumed

```
T = tuple(G) # (1, 9, 25, 49)
S = set(G) # {}
```

can write a fcn that returns a generator obj.

```
def odd_squares(n):
    return (x**2 for x in range(n) if x%2==1)
#end

G1 = odd_squares(8)
G2 = odd_squares(10)

T = tuple(G1) # (1, 9, 25, 49)
S = tuple(G2) # {1, 9, 25, 49, 81}
```

Functions that return generators can be much more complicated, logically. Use yield

```
def gen_fcn(...):
    ⋮
    yield exp
    ⋮
#end
```

← each execution of yield produces 1 term in the generator.

Ex. LinearSearch.py

Ex. Generators.py