# Generators

# Announcements

# Review: Iterables, Iterators

An iterable is any sequence we can iterate over (we can call **iter()** on it and get an iterator)

An iterator allows us to iterate over any iterable sequence (we can call **next()** on it and get the next item in the sequence)

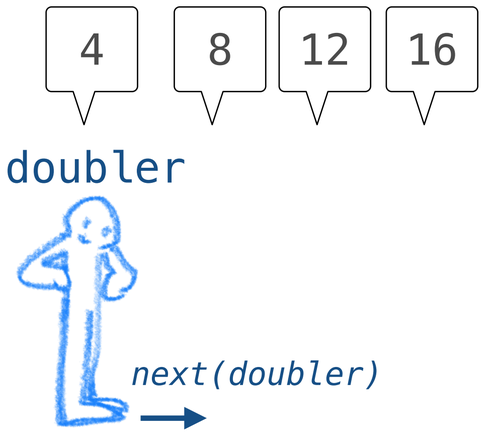
```
t = (1,2,3)
i = iter(t)
next(i)

l = ["John", "Jedi", "Shm"]
e = enumerate(l)
next(e)

d = {"apples": 1, "pears": 2}
m = map(lambda x: "yummy “ + x, d)
next(m)
```

other iterators:

zip(), filter(), reversed()

# Map Function Review

**map(func, iterable)** applies a given function to each item of an iterable




```
l = [2, 4, 6, 8]

doubler = map(double, l)
```

```
def double(x):
    print(f"*** doubling {x} ***")
    return x*2
```

# map() Practice

```
def add_to_each(p, edit):
    """
      Given a list, p, of 3-element tuples: [(x1, y1, z1), (x2, y2, z2), ...]
      And an edit tuple (also 3 elements) = (a, b, c),
      return a map object where
      a is added to each x-value,
      b to each y-value, and
      c to each z-value.

      >>> list(add_to_each([(0, 0, 0), (1, 1, 1)],  (10, 10, 10)))
          [(10, 10, 10), (11, 11, 11)]
      >>> list(add_to_each([(1, 2, 3), (1, 1, 1)],  (10, 20, 30))
          [(11, 22, 33), (11, 21, 31)]
    """
    return map(lambda x: ( x[0] + edit[0], x[1] + edit[1], x[2] + edit[2] ), p)
```

# Tree Practice

# Spring 2023 Midterm 2 Question 4(a)

Implement exclude, which takes a tree t and a value x. It returns a tree containing the root node of t as well as each non-root node of t with a label not equal to x. The parent of a node in the result is its nearest ancestor node that is not excluded.

```
def exclude(t, x):
    """Return a tree with the non-root nodes of tree t labeled anything but x.

    >>> t = tree(1, [tree(2, [tree(2), tree(3), tree(4)]), tree(5, [tree(1)])])
    >>> exclude(t, 2)
    [1, [3], [4], [5, [1]]]
    >>> exclude(t, 1)  # The root node cannot be excluded
    [1, [2, [2], [3], [4]], [5]]
    """
    filtered_branches = map(lambda y: ____________, branches(t))
    bs = []
    for b in filtered_branches:
        if ____________:
            bs.____(____)
        else:
            bs.append(b)
    return tree(label(t), bs)
```

exclude(y, x)

In Spring 2023, 20% of students got this right

label(b) == x

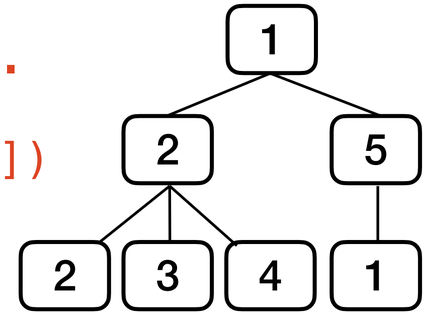
37% of students got this right

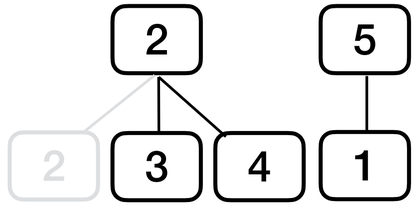
30% got it right; 1 of 4 options

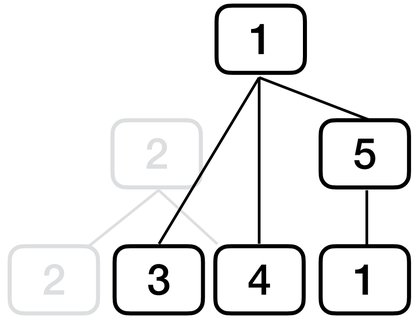
extend

branches(b)

24% got it right



# Generators

# Generators and Generator Functions

```
>>> def plus_minus(x):
...     yield x
...     yield -x

>>> t = plus_minus(3)
>>> next(t)
3
>>> next(t)
-3
>>> t
<generator object plus_minus ...>
```

A *generator function* is a function that **yield**s values instead of **return**ing them

A normal function **return**s once; a *generator function* can **yield** multiple times

A *generator* is an iterator created automatically by calling a *generator function*

When a *generator function* is called, it returns a *generator* that iterates over its yields

(Demo)

# Spring 2023 Midterm 2 Question 5(b)

**Definition.** When parking vehicles in a row, a motorcycle takes up 1 parking spot and a car takes up 2 adjacent parking spots. A string of length n can represent n adjacent parking spots using % for a motorcycle, <> for a car, and . for an empty spot.

For example: '.%%.<><>' (Thanks to the Berkeley Math Circle for introducing this question.)

Implement **park**, a generator function that yields all the ways, represented as strings, that vehicles can be parked in n adjacent parking spots for positive integer n.

```
def park(n):
    """Yield the ways to park cars and motorcycles in n adjacent spots.

    >>> sorted(park(1))
    ['%', '.']
    >>> sorted(park(2))
    ['%%', '%.', '.%', '..', '<>']
    >>> len(list(park(4)))  # some examples: '<><>', '.%%.', '%<>%', '%.<>'
    29
    """
```

# Fibonacci Generator

```
def fib_generator():
    """
    A generator that yields the Fibonacci sequence indefinitely.
    (The Fibonacci sequence starts with 0 and 1, and each subsequent number
    is the sum of the previous two.)

    >>> fib = fib_generator()
    >>> next(fib)
    0
    >>> next(fib)
    1
    >>> next(fib)
    1
    >>> next(fib)
    2
    >>> list(next(fib) for i in range(0,10)) # list the next 10 fibonacci numbers
    [3, 5, 8, 13, 21, 34, 55, 89, 144, 233]
    """
    a, b = 0, 1
    while True:
        yield a          # Yield the current Fibonacci number
        a, b = b, a + b  # Prepare the next Fibonacci numbers
```