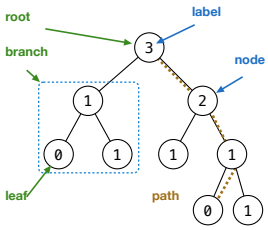
CS 61A Midterm 3 Study Guide

Trees

Recursive description:
- A **tree** has a root **label** and a list of **branches**
- Each branch is a **tree**
- A tree with zero branches is a **leaf**

Relative description:
- Each location is a **node**
- Each **node** has a **label**
- A node is a branch of its **parent**; its branches are its **children**



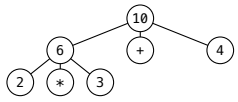

```
@dataclass
class Tree[T]:
    """A Tree has a label (of type T) and a list of branches."""
    label: T
    branches: list[Tree[T]] = field(default_factory=list)  # []
    def __str__(self):
        return format_tree(self)

def format_tree(t: Tree, indent='') -> str:
    "Format a tree with each branch indented below its label."
    assert isinstance(t, Tree) and isinstance(t.branches, list)
    string = indent + str(t.label)
    for b in t.branches:
        string += '\n' + format_tree(b, indent + '  ')
    return string

def is_leaf(t: Tree) -> bool:
    "Return whether a Tree t is a leaf with no branches."
    return not t.branches
```

```
>>> expr = Tree(10, [Tree(6, [Tree(2), Tree('*'),
...   Tree(3)]), Tree('+'), Tree(4)])
>>> expr.branches[0].label, len(expr.branches)
(6, 3)
```

```
>>> print(expr)
10
  6
    2
    *
    3
  +
  4
```




```
>>> expr.branches[1]
Tree(label='+', branches=[])
```

Each branch of a tree is a tree, so a function that takes trees can be called recursively on each of its branches.

```
def increment_leaves(t: Tree[int]) -> Tree[int]:
    "Return a tree like t with the leaf labels incremented."
    if not t.branches:
        return Tree(t.label + 1)
    incremented = [increment_leaves(b) for b in t.branches]
    return Tree(t.label, incremented)
```

List Mutation

```
>>> suits = ['coin', 'string', 'myriad', 'cup']
>>> suits.pop()                # remove & return last
'cup'
>>> suits.pop(1)               # remove & return index 1
'string'
>>> suits.remove('myriad')     # remove first match
>>> suits
['coin']
>>> suits.append('spade')      # add one to the end
>>> suits.extend(['heart', 'club'])  # add all to end
>>> suits[0] = 'diamond'
>>> suits.insert(0, 'joker')   # add at index
>>> suits
['joker', 'diamond', 'spade', 'heart', 'club']
>>> print(suits.append('x'))   # append returns None
None                           # so do extend, insert & remove
```

```
>>> s = [1, 2]
>>> s.append([3, 4])
>>> s.extend([5, 6])
>>> s
[1, 2, [3, 4], 5, 6]
```

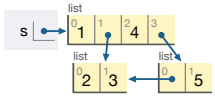
```
>>> t = s
>>> s += [7]      # mutates
>>> s = s + [8]   # new list
>>> t
[1, 2, [3, 4], 5, 6, 7]
```

```
>>> a = [10]
>>> b = a      # same list
>>> a == b
True
>>> b.append(20)
>>> a == b
True
>>> a
[10, 20]
>>> b
[10, 20]
>>> a is b
True
```

```
>>> a = [10]
>>> b = [10]   # new list
>>> a == b
True
>>> b.append(20)
>>> a == b
False
>>> a
[10]
>>> b
[10, 20]
>>> a is b
False
```

Mutation and Box-and-Pointer Diagrams

```
>>> s = [1, [2, 3], 4]
>>> s.append([s[1], 5])
>>> print(s)
[1, [2, 3], 4, [[2, 3], 5]]
```



Dictionary Mutation

```
>>> words = {"oruguita": "caterpillar"}
>>> words["oruguita"] += "🐛"
>>> words["oruguita"]
'caterpillar🐛'
>>> words["mariposa"] = "butterfly"
>>> words
{'oruguita': 'caterpillar🐛', 'mariposa': 'butterfly'}
```

Copies and Tuples

Making a new list does not copy nested lists it contains.

```
>>> a = [[1], 2]
>>> c = list(a)
>>> c[0].append(5)
>>> c[1] = 9
>>> a
[[1, 5], 2]
>>> (c is a, c[0] is a[0])
(False, True)
```

A tuple can't change, but a list inside it can.

```
>>> t = (1, [2])
>>> t.append(3) # no append
AttributeError
>>> t[1].append(3)
>>> t
(1, [2, 3])
```

Iterators & Iterables

An **iterable** value is one that **for** can iterate over (list, tuple, range, dict, str). An **iterator** provides the items of an iterable one at a time, tracking its position through the iterable. Iterators are also iterables.

`iter(iterable)` make an iterator over the iterable's items
`next(iterator)` return the next item in the iterator

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
4
```

```
>>> d = {'one': 1,
...      'two': 2}
>>> k = iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
```

```
>>> v = iter(
...   d.values())
>>> next(v)
1
>>> next(v)
2
```

Once an iterator reaches the end, calling `next` on it causes an error (`StopIteration`). Calling `iter` again on an iterable such as a list makes a new iterator.

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
4
```

```
>>> u = iter(s)
>>> next(u)
3
>>> next(t)
5
>>> next(u)
4
```

- A **for** statement calls `iter`, then `next` until StopIteration.
- `iter` of an iterator returns itself.
- Adding a dict key while iterating over it: RuntimeError.

Lazy Evaluation

These functions return iterators whose evaluation is lazy:

`map(func, iterable)`	func(x) for x in iterable
`filter(func, iterable)`	x in iterable if func(x)
`zip(first, second)`	co-indexed (x, y) pairs
`reversed(sequence)`	x in reverse order

To view the contents of an iterator, place the items into a **container** with `list`, `tuple`, or `sorted`.

```
def double(x):
    print('***', x, '=>', 2*x, '***')
    return 2*x
```

```
>>> s = [3, 4, 5]
>>> doubled = map(double, s)
>>> next(doubled)
*** 3 => 6 ***
6
```

```
>>> list(doubled)
*** 4 => 8 ***
*** 5 => 10 ***
[8, 10]
```

Generators

A **generator function** has **yield** in its body. Calling it **doesn't run the body**: it returns a **generator**, which is an iterator. Each `next` runs the body until the next **yield**, returns that value, and **pauses** there (local names keep their values). When the body ends, `next` raises StopIteration.

```
def countdown(n):
    print('start')
    while n > 0:
        yield n
        n -= 1
    print('done')
```

```
>>> g = countdown(2)
>>> next(g)
start
2
>>> next(g)
1
>>> next(g)
done
StopIteration
```

countdown(2) printed nothing; each `next` resumes after **yield**.

yield from <it> yields every value of <it>, one at a time.

```
def a_then_b(a, b):
    yield from a
    yield from b
```

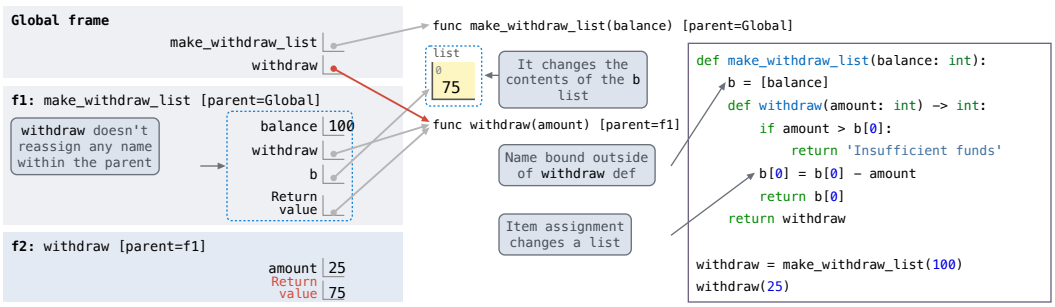
```
>>> list(a_then_b([3, 4],
...               range(5, 7)))
[3, 4, 5, 6]
```

A recursive call to a generator function returns an iterator.

```
def prefixes(s):
  if s:
    yield from prefixes(s[:-1])
    yield s
```

```
>>> list(prefixes('abc'))
['a', 'ab', 'abc']
```

Environment diagram: mutation inside a function



Python Object System

When a class is called:
1. A new instance of that class is created.
2. The `__init__` method of the class is called with the new object as its first argument (named `self`), along with any additional arguments provided in the call expression.

Example: All bank accounts have a **balance** and a **holder**; the **Account** class adds those attributes to each of its **instances**.

```
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

A new instance is created by calling a class

An account instance: balance: 0 holder: 'Jim'

```
class Account:
    interest = 0.02   # class attribute: shared
    def __init__(self, account_holder):
        self.balance = 0     # instance attributes:
        self.holder = account_holder  # one per object
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
```

`__init__` is called a **constructor**

self should always be bound to an instance of the Account class or a subclass of Account

```
>>> type(Account.deposit)
<class 'function'>
>>> type(a.deposit)
<class 'method'>
```

Looked up by the class: a plain **function**. Looked up by an instance: a **bound method**

```
>>> Account.deposit(a, 10)
10
```

Function call: all arguments within parentheses

```
>>> a.deposit(2)
12
```

Method invocation: one object before the dot, other arguments within parentheses

```
>>> f = a.deposit
>>> f(10)
22
>>> a.balance
22
>>> a.balance()      # int not callable
TypeError
```

Class vs. Instance Attributes

Assigning to <expression>.<name> sets an attribute on that object: on an **instance**, an instance attribute; on a **class**, a class attribute.

```
>>> jim = Account('Jim')
>>> tom = Account('Tom')
>>> tom.interest
0.02
>>> Account.interest = 0.04
>>> tom.interest
0.04
>>> jim.interest
0.04
```

```
>>> jim.interest = 0.08
>>> jim.interest
0.08
>>> tom.interest
0.04
>>> Account.interest = 0.05
>>> tom.interest
0.05
>>> jim.interest
0.08
```

Account (class)	jim (instance)	tom (instance)
interest: ~~0.02~~ ~~0.04~~ 0.05	balance: 0 interest: 0.08	balance: 0

Inheritance

A **subclass** inherits all attributes of its **base class** and can **override** some. Use it for **is-a** relationships (a checking account *is an* account).

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return super().withdraw(amount + self.withdraw_fee)
```

super() = "the parent class's version, bound to self". Inside `CheckingAccount.withdraw`, `super().withdraw(x)` skips `CheckingAccount` and looks `withdraw` up starting in `Account`, so the **original** behavior is used. `self` is passed automatically.
- Same as `Account.withdraw(self, x)`.
- **Not** `self.withdraw(x)`; that finds `CheckingAccount.withdraw` which results in infinite recursion.
- Write `self.withdraw_fee`, not `CheckingAccount.withdraw_fee`, so a subclass or instance can override the fee.
- A subclass `__init__` **overrides** the parent's; call `super().__init__(...)` to also run the parent's setup.

```
>>> ch = CheckingAccount('Tom')  # Account.__init__
>>> ch.interest     # Found in CheckingAccount
0.01
>>> ch.deposit(20)  # Found in Account
20
>>> ch.withdraw(5)  # Found in CheckingAccount
14
```

Looking up obj.name:
1. Evaluate `obj` to get an object.
2. If `name` is an **instance attribute** of that object, use it.
3. Otherwise look in the object's **class**, then its **base class**, then that class's base class, and so on.
4. Not found anywhere: AttributeError.

If an attribute is found in the class and is a function, you get a **bound method.**

`obj.name = v` always sets the attribute `name` on `obj`.

Memoization

Save each result (e.g. in a dict keyed by the argument) so a repeated call is just a lookup instead of being recomputed. A memoized tree-recursive `fib` computes each fib(k) only once.