Prep: bring ID, watch, a pencil (for environment diagram), lots of paper, pen, water and this
You got this!

WWPD

`>>> "hi"`
`'hi'`
`>>> 'hi'`
`'hi'`
`>>> 1 == True`
`True`
`>>> 0 == False`
`True`
	`>>> repr([1, 2, 3])`
	`'[1, 2, 3]'`
	`>>> x = {}`
	`>>> x["jz"] = "rn"`
	`>>> repr(x)`
	`"{'jz':'rn'}"`

`>>> repr("jianzhi")`
`"'jianzhi'"`
`>>> str("jianzhi")`
`'jianzhi'`
`>>> "jianzhi"`
`'jianzhi'`
	`>>> repr(2)`
	`'2'`
	`>>> repr(None)`
	`'None'`
	`>>> str(None)`
	`'None'`

Tree, Link

- Link

Class Var	Constructor
`Link.empty`	`l = Link(1, Link(2))`
Instance Var	**Methods**
`l.first` `l.rest # must be Link instance`	

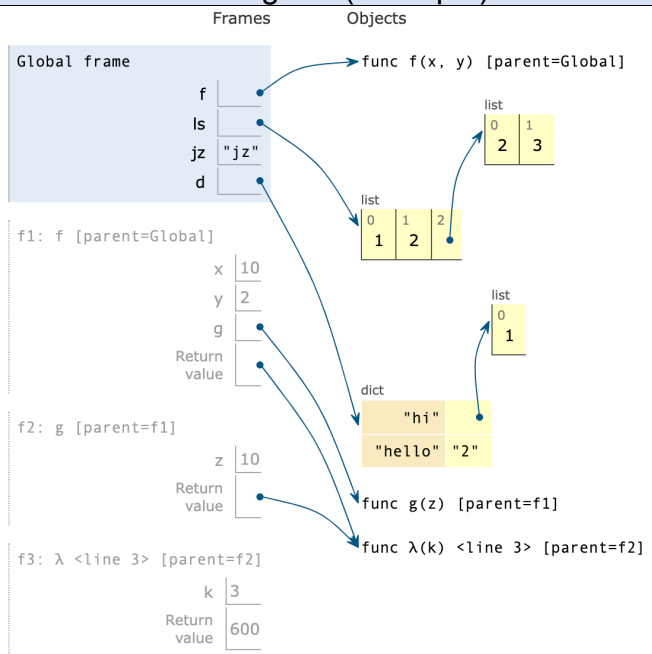
- Tree

Class Var	Constructor

`None`	`t = Tree(label, branches=[])`
	
	`t = Tree(2)`

Instance Var	**Methods**
`t.label` `t.branches`	`t.is_leaf() # FUNCTION`

Environmental Diagram (Example)



Data Structures

1. List

Method	Remarks
`ls.append(x)`	Returns `None`
`ls.insert(1, "a")`	Returns `None`
`ls.pop()`	Returns `<element>`
`ls.pop(1)`	Returns `<element>`
`ls.reverse()`	Returns `None`
`ls.remove(x)`	Returns `None`
`ls.extend([1])`	Returns `None`
`ls.copy()`	Returns new list
`ls.index(x)`	Returns index of `x`
`ls.count(x)`	Returns number of `x`
`ls[:]`	Returns new list

List Comprehensions

```
[x for x in range(1, 12) if 12 % x == 0]
[x if 12 % x == 0 else 1 for x in range(1, n)]
```

Shallow Copy

`list(ls)`	`[ls[0], ls[1], …, ls[n]]`	
`ls + []`	`ls[:]`	`ls.copy()`

2. String

`" ".join(<iterable>)`
`" ".join(["b", "luv", "z"]) # delimiter`
`s.split("e") # split by delimiter into list`
`s.split() # default " "`

3. Dictionary

Method	Return
`d.get("x", 0)`	`d[x] if "x" in d else 0`
`d.pop("key")`	`d["key"] else KeyError`
`list(d.keys())`	list of keys
`list(d.items())`	list of `(key, item)`

`dd = d.copy()`	returns a shallow copy
	`dd == d # True`
	`dd is d # False`

`d.clear()`	clears the dictionary
`d.popitem()`	returns the item last inserted and removes it
`d.setdefault("x", v)`	returns `d["x"]` if `x` in `d`, else return `d["x"] = v`

Iteration

`for i in d.keys():` `  print(i)`	`for k, v in d.items():` `  print(k, v)`
`iter(prices.keys())`	`iter(prices.items())`

Dictionary Comprehensions

```
x = {i : s[i] for i in range(len(s)) if s[i]
!= w}
```

Iterators and Generators

Iterables: `[], (), "string", range(1, 21), {"dict": 1}`

Iterable Functions:

- `list(<iterable>)`
- `type(<iterable>) # == tuple, list, dict, str`
- `sorted(<iterable>)`

```
if (type([1, 2, 3]) == list):
    # logic
```

Iterator Functions:

- `iter(<iterable>)`
- `iter(<iterator>)` returns itself
- `next(<iterator>)`
- `list(<iterator>) # exhausts the iterator`

```
for i in <iterator>: # python calls next()
    # logic
```

If called on an iterator, will exhaust it.
`StopIteration exception`

Generator:

- Key words: `yield`, `yield from`
- Usually comes with `while` loop
- If you do a `yield from`, remember to `return`

```
def gen(x):
  while x > 0:
    yield x
    x -= 1
```

```
def g(x):
  while x > 0:
    yield x
    yield from g(x//2)
    return
```

Generator Functions:

- `yield from <iterable>`
- `yield from <generator>`
- `list(<generator_with_args>)`

```
for i in <generator_function>:
    # logic
```

Functions on Iterable

```
list(<iterable>)
list(<iterator>)
tuple(<iterable>)
tuple(<iterator>)
```

- reversed(<iterable>)

```
<reversed object at 0x7fe37836dc70>
# same as reversed iterator
it = reversed(["jz", "doing", "cs61a"])
next(it) # 'cs61a'
```

- all(<iterable>)

```
all(<iterable>) # returns a boolean
all([]) == True
```

- any(<iterable>)

```
any(<iterable>) # returns a boolean
any([]) == False
```

- max(<iterable>) / min(<iterable>)

```
max(["jz", "hi", "wang"]) = "wang" #
lexigraphic
max([]) # ValueError
max(["jianzhi", "hi", "wang"], key = lambda x:
len(x)) # "jianzhi"
```

- sum(<iterable>, <initial value>)

```
sum([]) == 0
sum([1, 2, 3]) == 6
>>> sum([[x] for x in [1, 2, 3]], [])
[1, 2, 3]
>>> sum([[[x]] for x in [1, 2, 3]], [])
[[1], [2], [3]]
sum([[t.label + x for x in f(y)] for y in
z], []) # double list summation
```

- string
  - `text.lower()`
  - `text.upper()`
  - `text.split()`

OOP and Inheritance

Method Resolution Order (MRO)

```
class A(B, C):
    # logic
    # searches A then B then C
```

Calling super's methods:

```
super().__init__(title, author)
super().__str__()
# calls the __str__(self) in super class
```

```
class Car:
    wheels = 4
    def __init__(self, name):
        self.name = name
        self.fuel = 0
    def add_fuel(self, x):
        self.fuel += x
```

__Dunder__()

isinstance, dir, __dict__, magic methods, __repr__, __str__, fstrings

`str()` and `repr()`

```
>>> repr(x)
'<probably with quotes>'
```

`repr()` calls `__repr__()`, returns a **string**

```
>>> print(obj)
<str(obj) - probably no quotes>
```

`print(obj)` implicitly calls `str()` on it, and then removes one layer of quotes

```
x = Car("Tesla")
x.add_fuel(10) # equiv to Car.add_fuel(x, 10)

class FastCar(Car):
    def __init__(self, name, brand):
        super().__init__(name)
        self.brand = brand

    def add_fuel(self, x):
        super().add_fuel(2*x)
        # equiv to Car.add_fuel(self, 2*x)

Car.wheel #4
x.wheel #4
```

CANNOT to modify class var through instance

```
>>> getattr(obj, "name") # returns value or
AttributeError
>>> hasattr(obj, "name") # returns boolean
```

```
>>> obj
<repr(obj) _ probably no quotes>
```

Python calls `repr()` method on it, obtain `res`, calls print on `res`, which removes quotes.

```
>>> str("a")
'a'
```

Just strings

```
# called during print(), which removes quotes
# called during str()
# if not defined, defaults to __repr__
def __str__(self):
    return f'hi'

# called during >>> obj
def __repr__(self):
    return f"PaperReam('{self.x}')"

def __iter__(self):
    return iter(self.leds)

def __iter__(self):
    for led in self.leds:
        yield led
```

Misc

Range (exclusive)

- `range(start=0, stop, step=1)`
- `range(start, stop)`
- `range(stop)`

zip(*iterables)

```
>>> zip([1, 2, 3], [7, 8, 9]) # iterator
<zip object at 0x7f810aa4f440>
for i, j in x: # next(x) returns tuple
    print(i, j)
```

map(func, iterable, …)

```
map(lambda x: x*2, [1, 2, 3])
# returns iterator on resultant list
print(map(lambda x: x*2, [1, 2, 3]))
<map object at 0x7fa31936dc70>
list(map(lambda x: x*2, [1, 2, 3]))
```

filter(func, iterable)

```
filter(lambda x: x % 2 == 0, [1, 2, 3, 4])
<filter object at 0x7fdb8bc52c10>
# returns iterator on resultant list
list(filter(lambda x: n%x==0, range(1, n+1)))
```

reduce(lambda, iterable)

```
from functools import reduce
total = reduce(lambda a, b: a+b, [1, 2, 3, 4])
```

- fstring (Do NOT mix ‘ and “)

```
return f'Current {self.item} stock:
{self.stock}'
```

Last Resorts

1. Python ternary:

```
return x if <condition> else y
```

2. Tuple assignment:

```
x, y = y, x + y
```

3. Chaining assignment (just use tuple)

```
L = L.rest = L.rest.rest
```

Line of Attack

- Eye power
- Iteration, recursion
- Get hints from previous functions in the same problem; Inspect arguments & return value
- Exploit other values
- Use lambda functions, double lambda; Remember that functions can take on values also
- Look at lambda variants below
- Recursion (`__str__` is sort of recursive)
- Whack, try test cases, whatever works
- Check and verify examples

λ Functions

- 2 Recursive

```
def d(f):
    def g(i):
        # logic
        # calls parent function with args
        # i == x is memorized
```

Final Checks

- **`n // 10` VS** `n / 10`
- `float("-inf")`
- `pow()` returns float when any argument is float, use `round()`
- `range(x, y)` is exclusive

```
        return (lambda x: f(x) or i == x)
    return g
```

- 3-recursive (typically consists of initiation function, then a 2-recursive)

```
def maxer(smoke):
    # initiation function
    def fire(y):
        # 2-recursive parent
        def haze(z):
            # 2-recursize child
            # logic
            return fire(z);
        return haze
    return fire

def compose(n):
    # returns g(x) = f1(f2(…fn(x)…))
    if n == 1:
        return lambda f: f
    def call(f):
        def on(g):
            return compose(n – 1)(lambda x:
f(g(x)))
        return on
    return call
```

- Multiple lambdas

```
# Fall 2016
def multiadder(n):
       # >>> multiadder(3)(5)(6)(7)
       # 18
       if n == 1:
              return lambda x: x
       else:
              # essentially lambda, since one
argument given
              return lambda x: lambda y:
multiadder(n – 1)(x + y)

zipper = lambda x: x
helper = lambda f, g: lambda x: f(g(x))
zipper = helper(f1, zipper)
```

- Check base cases (`[[]]` in 19 Summer)
- **`self.`**`var`
- Read doc tests and problem statement
- Check all methods and variables of objects