CS 61A = Death

Fall 2019

MW 2-3 Pimentel 1 Lec

Tu 9:30-11 Soda 277

Th 11-12:30 Barrows 185

8/28/19 Lecture

What is CS?

A course of managing abstraction
- mastering abstraction
- programming paradigms

A intro to programming
- full understanding of Python fundamentals
- how computers interpret programming

Different types of languages: Scheme & SQL

- look @ announcement
- Parts of Course:
  - watch video before lecture!
  - lab is most important!
  - discussion most important!
  - office hrs most important!
  - read textbook before class!
  - look @ important dates!
  - cs61a.org

Expressions: describes a computation and evaluates a value

f(x) → most overall   max(3, 4.5)   pow(2, 100)

Course Policy:
- discuss everything!
- completed with partner
- partner from discussion!
- build good habits now!

Anatomy of Call Expression

$\underbrace{\text{add}}_{\text{operator}}(\underbrace{2}_{\text{operand}}, \underbrace{3}_{\text{operand}})$

28   8

mul(add(4, mul(4,6)), add(3,5))   mul(28, 8) → 224

mul

28

add(4, mul(4,6))   add(3,5)

8

add   4   24

mul(4,6)

mul   4   6

- open python? python3

Directory Tools:
- ls: list files in current directory
- cd: changes to specific directory
- mkdir: makes new directory
- mv <source path> <destination path>: move source to new area

7 % 2 → remainder of division

7 // 2 → floor rounding division

8/29/19 Discussion

lilliandu@berkeley.edu
- turn day before due date for projects
- always go to class and lecture

8/30/19 - Lecture : Names

Announcements
- HW 1 due
- lab on tuesday!
- lecture 3 is video only (look on website)
- lecture 4 : 1 Pimentel, 150 Wheeler

Names, Assignment, User-Defined Functions

control L → disappear

$>>>$ → python prompt

max (2,3) : 3

pow (2,3) = 8

* import names!

import names :

$>>>$ from math import pi

$>>>$ from math import sin

assignment :

$>>>$ radius = 10

* once number is bound to value, will continue staying bound

$>>>$ area, circ = pi * radius * radius, 2 * pi * radius

max(2,3) → call function

f = max → f (2,3) = 3 also

↳ set f to max function

if place max = 7, then max becomes int, not function anymore

* need to exit → exit() & control D

```
def square(x):
... return mul(x, x)
def sum_squares(x,y):
... return sum(square(x), square(y))
```

square (3)

↳ 9

* from math import add

* can make area a function to show updated number every time, remembers expression not the number

```
def area():
... return pi * radius * radius
```

→ but would need to define radius before or it errors

f = max g = min h = max max = min

max (f (2, g (h (1,5), 3)), 4)

max (f (2, g (5 , 3), 3)), 4)

max (f (2 , 3 , 3), 4)

max (3 , 4) → $\boxed{3}$

* make expression tree!

Environment Diagrams

visualize interpreter's process

pi | 3.1415

f = min()

↳ would try to compute the function

f = min

↳ would straight set the value

Assignment Statements:

		global frames
1	a = 1	a | 1
2	b = 2	b | 2
3	b, a = a + b, b	a | 2
		b | 3

Execution rule for assignment statements:

1. evaluate all expressions to the right of = from left to right
2. bind all names to the left of = to the resulting values in the current frame

Defining Functions

Assignment is a simple means of abstraction: binds name to value

Function definition is a more powerful means of abstraction: binds name to expression

def <name>(<formal parameters>): → signature: # of arguments taken

return <return expression> → return body: computational process expressed by a function

Execution procedure for def statements:

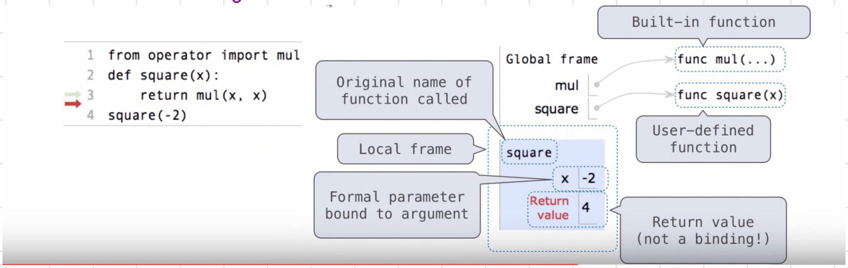
Procedure for calling | applying user defined functions

1. creates a function with signature <name> (<formal parameters>)
2. set the body of that function to be everything indented after the first line
3. bind <name> to that function in the current frame

Calling User-Defined Functions

Procedure for calling | applying used defined functions

1. Add a local frame, forming a local environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment.



* function's sig has all info needed to create a local frame

Looking Up Names in Environment

Every expression is evaluated in the context of an environment.
So far, the current environment is either:
- the global frame alone, or
- a local frame, followed by the global frame

* Important!

An environment is a sequence of frames

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

- first looks for name in that local frame
- if not found, look for it in global frame

9/1/16 Lecture - Control

Announcements

- code.cs61a.org
-

Print vs. None

- python has rules to automatically print what evaluated
- "None" represents none in python
- function will return 'none' if nothing explicitly returned
- None not displayed as value of expression

```
def does_not_square(x):
...   x * x          no return
>>> does_not_square(4)
```

~~None~~ → can't display none as expression value

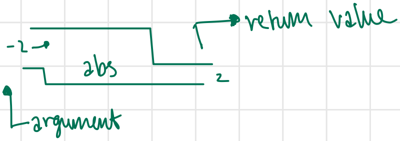
```
>>> sixteen = does_not_square(4)
```

sixteen = NONE

Pure Functions & Non-Pure Functions

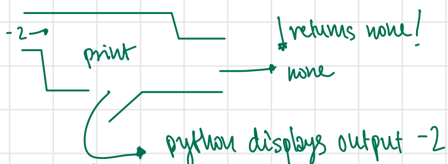
Pure Functions

just return value



Non-Pure Functions

have side effects



A side effect isn't a value; just something that happens as a consequence of calling a function

print (print (1), print (2))
$\underbrace{}_{1}$ $\underbrace{}_{2}$

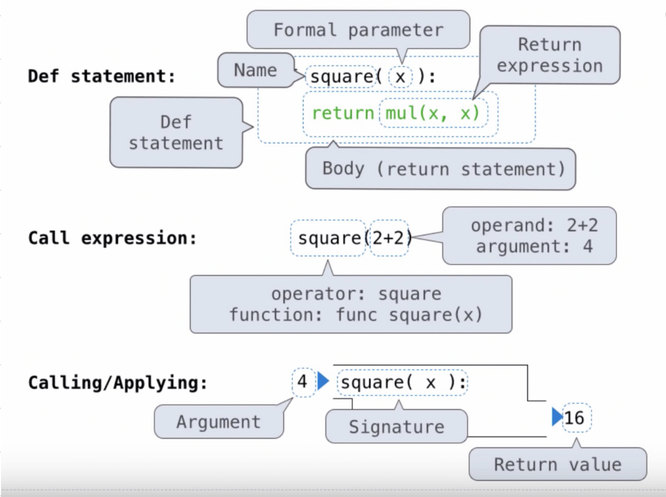
print ((doesn't have a return), (doesn't have a return))

1

2

None None

Life Cycle of a User-Defined Function



What happens?

A new function is created!
Name bound to that function in the current frame

Operator & operands evaluated
Function (value of operator) called on arguments (values of operands)

A new frame is created!
Parameters bound to arguments
Body is executed in that new environment

example code:
1 | from operator import mul
2 | def square(x):
3 | return mul(x, x)
4 | square (square(3))

global frame
mul | → func mul(...)
square | → func square(x)

f1: square [parent: global]
x | 3
return value | 9

f2: square [parent: global]
x | 9
return value | 81

An environment is a sequence of frames.
- global frame alone
- a local, then the global frame

Every expression is evaluated in the context of an environment.
A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

Misc Python Features

division:
- 2013/10 → 201.3 (true divide) (truediv)
- 2013//10 → 201 (integer division) (floordiv)
- 2013 % 10 → 3 (remainder div) (mod)

importing:
python3 <whatever name>.py
""" comments ♡ """

doctest

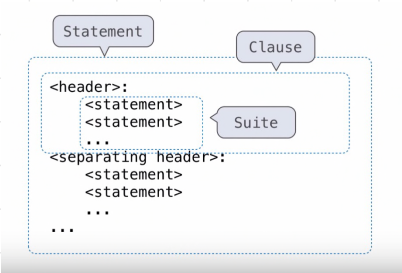
```
def divide_exact(n, d):
    """Return the quotient and remainder of dividing N by D.

    >>> q, r = divide_exact(2013, 10)
    >>> q
    201
    >>> r
    3
    """
```

-m doctest -v ex.py
→ way to check if right by using """ and setting what vars supposed to be
* if value not given in parameter, automatically uses doctest value

Conditional Statements

A statement is executed by the interpreter to perform an action.

Compound Statement:



The first header determines a statement's type.

The header of a clause "controls" the suite that follows.

def statements are compound statements

To "execute" a suite means to execute its sequence of statements, in order

Execution Rule for a sequence of statements

- execute the first statement
- unless directed otherwise, execute the rest

ex.

```
def absolute_value(x):
    if x < 0:
        return -x
    elif x == 0:
        return 0
    else:
        return x
```

2 boolean contexts

-i ex.py

to open a python wordedit

Syntax Tips

1. Always starts with "if" clause
2. 0 or more "elif" clauses
3. 0/1 "else" clauses, always in end

Execution rule for conditional statements

Each clause is considered in order

1. Evaluate the header's expression
2. If it is a true value, execute the suite & skip the remaining clauses

Iteration

While:

```
1  i, total = 0, 0
2  while i < 3:
3      i = i + 1
4      total = total + i
```

Execution Rule for while:

1. evaluate the header's expression
2. if it is a true value, execute the (whole) suit, return to step 1

Lab 1 Crashcourse Notes

Division

True Division: / (decimal division)	Floor Division: // (integer division)	Modulo: % (remainder)
> 1/5	> 1//5	> 1 % 5
0.2	0	1
> 25/4	> 25//4	> 25 % 4
6.25	6	1
> 4/2	> 4//2	> 4 % 2
2.0	2	0
> 5/0	> 5//0	> 5 % 0
ZeroDivisionError	ZeroDivisionError	ZeroDivisionError

* useful technique involving % operator : check if # divisible

x % y == 0 — x divisble by y

x % 2 == 0 — x divisble by 2

Functions

can make function to abstract a line of stuff

```
def foo(x):
    return x * 3 + 2
```

```
> foo(1)
5
```

applying function is done with call expression

Caller Expressions

add (2, 3)

operator — operand

Evaluate a function:

1. Evaluate the operator, and then the operands (from left to right)
2. Apply the operator to operands (the values of operands)

* if nested expression, apply to inner operand then outer operand

Return vs. Print

- if executes return statement, then function terminates immediately
- if reaches end of function body w/out return → returns None
- print just prints in Terminal, w/out terminating

* print displays text without quotes, return will preserve quotes

Control

Boolean Operators

- and evaluates to true only if both operands evaluate to true
  - ↳ atleast one is false then evaluates to false
- or evaluates to True if atleast one operand is true
  - ↳ all are false then evaluates to false
- not evaluates to true if operand evaluates to false
  - ↳ evaluates to false if operand evaluates to true

ex. True and not False or not True and False

True & True or False and False

True or False

Order of Operations for Boolean

- not → highest priority
- and
- or → lowest priority

Short Circuiting:

Operator:	Checks if:	Evaluates left to right:	Example:
and	all values true	first false value	False & 1/0 → False
or	atleast one true	first true value	true or 1/0 → true

* if and & or don't short-circuit, then they return last value

If-Statements:

```
if x > 3:
    return True
else:
    return False
```

While Loops:

```
while (blah):
    do blah
```

9/4/19 - Lecture: High-Order Function

Announcements
- HW 1 due thurs 9/5
- Hog due thurs 9/11
  - Checkpoint due 9/9
- additional topics lecture: 9/4 5-6 wed 3106 Etche

Iteration Example

The Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8

index: 0th 1st 2nd 3rd 4th 5th 6th

```
def fib(n):
    pred, curr = 0, 1      # 0th and 1st Fib #s
    k = 1                  # curr is kth Fib #
    while k<n:
        pred, curr = curr, pred + curr
        k = k+1
    return curr
```

pred, curr = 1, 0
k=0

also takes into account 0th value!

* for assignment statements, right always done before left

fib pred
curr
n
k

Environment Diagram

GF
fib → func fib(n) p=g
pred ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ 3
curr ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ 5
k ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ 5
rv 5

GF
fib → func fib(n) p=g
pred 0 ~~1~~ ~~1~~ 2
curr ~~1~~ ~~1~~ ~~2~~ 3
k ~~1~~ ~~2~~ ~~3~~ 4
rv 3

Designing Functions

- A function's domain is the set of all inputs it might possibly take as arguments
- A function's range is the set of output values it might possibly return
- A pure function's behavior is the relationship it creates between input and output

Def square(x)
D: x is real number
R: non-neg real number
RV: square of input

Def Fib(n)
D: x is real number
R: returns fib number
RV: nth fib number

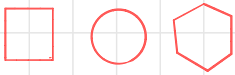
A Guide to Design Function

- Give each function exactly one job
- Don't repeat yourself (DRY). Implement a process just once, but execute many times
- Define functions generally

Higher-Order Functions

Generalize Patterns with Arguments

Regular geometric shapes relate length and area

Shapes: 

Area: $1 \cdot r^2$ $\pi \cdot r^2$ $\frac{3\sqrt{3}}{2} \cdot r^2$

Finding common structure allows for shared implementation

```
def area_square(r):
    return r*r
def area_circle(r)
    return r*r*pi
def area_hexagon(r)
    return r*r*3*sqrt(3)/2
```

repeating ourselves w/ the assert statement & return r*r part!

```
assert 2>3, 'Math is Broken'
```

→ allows to test for certain conditions that aren't allowed

```
def area(r, shape constant)
    assert r>0, 'A length must be positive'
    return r*r*shape constant
def area_square (r)
    return area (r, 1)
def area_circle (r)
    return area (r, pi)
def area_hexagon (r)
    return area (r, 3*sqrt(3)/2)
```

Generalizing Over Computational Process

The common structure among functions may be a computational process, rather than a number

$$\sum_{k=1}^{5} k = 1+2+3+4+5 = 15$$

$$\sum_{k=1}^{5} k^3 = 1^3+2^3+3^3+4^3+5^3 = 225$$

$$\sum_{k=1}^{5} \frac{8}{(4k-3)(4k-1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} = 3.04$$

```python
def sum_naturals(n):
    """ Sums first n natural numbers
    >>> sum_naturals(5)
    15
    """
    total, k = 0, 1
    while k <= n
        total = total + k
        k = k + 1
    return total
```

```python
def sum_cubes(n):
    total, k = 0, 1
    while k <= n:
        total, k = pow(k, 3), k+1
    return total
```

```python
def pi_term(k):
    return 8 / mul(4*k-3, 4*k-1)
```

```python
def identity(k):
    return k
def cube(k):
    return pow(k, 3)
def summation(n, term):
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k+1
    return total
def sum_naturals(n):
    return summation(n, identity)
def sum_cubes(n):
    return summation(n, cube)
```

```python
def make_adder(n):
    "Return a function that take k
     and returns k+n"
    def adder(k):
        return k+n
    return adder
```

Function as a return value

f = make_adder(2000)(13)

→ make_adder(2000)
adder(13)

or f = make adder(2000)
f(13) = 2013

make_adder(1) (2)

func adder(k) — make_adder(1) — 2

func make_adder(n) — 1 — adder(2)

The Purpose of High-Order Function

Functions are first class: Functions can be manipulated as values in our programming language

Higher-Order Functions: A function that takes a function as an argument value or returns a function as a return value

Higher-Order functions:

- Express general methods of computation
- Remove repitition from programs
- Separate concerns among functions

Lamda Expressions

square = lambda x : x * x

square (4) = 16

square (10) = 100

(lambda x: x * x , 10)

A function

with formal parameter x

that returns value of "x * x"

Lambda cannot use def statements

Lambda vs. Def Statements

square = lambda x: x * x

def square(x):
    return x * x

- Both create a function with the same domain, range, and behavior
- Both functions have as their parent the function in which they were defined
- Both bind that function to the name square
- Only the def statement gives the function an intrinsic name

* makes weird environment diagrams

Gf

square | → func λ(x) p=g

f1: λ (p=g)

x | 4

rv | 16

use lambda, not actual name

Gf

square | → func square(x) p=g

f1: square (p=g)

x | 4

rv | 16

Control Structure Practice Problems

Hog Rules

- Pig out → any are 1 → score for turn is 1
- Free Bacon → if chooses 0 rolls, then you do 10- min between ten's & one's digit
- Feral hogs → if dice you roll is exactly 2 away from last one, get 3 extra points (absolute dif)
- Swine Swap → if left & right most digits swapped = opponent's left & right → scores are swapped

Problem 1:

a = even ; odd ; 3
b = one ; three ; 25
c = two ; four ; 8
d = odd ; 4
even ; odd ; 3
e = two ; four ; 8
one ; four ; 13

since it only set the variables, only prints the print statements, not the return

Problem 2:

```
temp1 = 0
while n:
    k = n % 10
    if (temp1 > k):
        return False
    n = n // 10
    temp1 = k
return True
```

2 → 4
1 → 6 2 + 2 + 3 + 2

Q. unlocked case # 6 → remembers where it last stopped

announce highest (who, previous high, previous score)
f0 = announce (c)
f1 = f0 (12,

Problem 3:

```
count = 0
while n:
    k = n % 10
    if (k = 1):
        count += 1
    n = n // 10
return count
```

Problem 4:

```
count = 0
while (n != 1)
    count = count + count_one(n)
    n = n - 1
return count
```

9/5/19 Discussion Notes

- Expressions vs. Value (Q vs. A)
- Everything has T & F value
  - True: True, 1, 10, -10, "hello"
  - False: False, 0, " ", None, print("hello")
- And: evaluates left to right, return first False, or return last True
- Or: return first True, or return last false evaluated

Warm-Up

(True or 10) and 'hello' and (-9 and 5-5)

① True ② "hello" ③ 0

True and "hello" and 0

→ returns 0, not False!

Print vs. Return Statement

```
def f():
    return "hello"
def g():
    print("hello")
```

* call function → need f()

if just f, will show where function is

x = f() → returns nothing   x → "print"

y = g() → prints hello   y → shows nothing

Control

- repeat chunks of code

- infinite loop → won't ever tell you if errored

Environment Diagrams

- always start with the global frame
- if function never included return, you don't need to add them either
- operand should always be a value
- parent of function is where it's defined, not called
- don't leave any brackets blank
- don't open new frame for built-in functions

9/6/19 - Lecture: Environments

Higher-order function: a function that takes a function as an argument value or returns a function as a return value

```
def apply_twice (f, x)
    return f(f(x))
def square (x):
    return x * x
result apply_twice (square, 2)
```

global
apply_twice → func apply_twice (f, x) p:g
square → func square(x) p=g
result | 16

f1 apply_twice(square, 2) p=g
f | square
x | 2
rv | 16

f2 square (x) p=g
x | 2
rv | 4

f3 square (x) p=g
x | 4
rv | 16

Applying function
- create a new frame
- bind formal parameters (f ; x) to arguments
- execute the body: return f (f(x))

global
repeat → func repeat(g, 5) p=g
g → func g (y) p=g

f1 repeat (g, 5) p=g
f | g
x | ~~5~~ ~~3~~ 2
rv | 2

g=f

f4 g(y) p=g
y | 2
rv | 2

- nested functions are in parent of their bigger function

f2 g(y) p=g
y | 5
rv | 3

f3 g(y) P=g
y | 3
rv | 2

```
def make_adder (n):
    def adder(k):
        return k+n
    return adder
```

global frame
make_adder → func make_adder(n) p=g
add_three →
result | 7

f1: make_adder(n) p=g
n | 3
adder → func adder (k) p=f1
rv →

f2: adder (k) p=f1
k | 4
rv | 7

- Every user-defined function has a parent frame (often global)
- the parent of a function is the frame in which it was defined
- every local frame has a parent frame (often global)
- the parent of a frame is the parent of the function called

How to Draw an Environment Diagram

When function is defined:

create a function value: func <name> (<formal parameters>) [parent= <parent>]

Its parent is the current frame.

Bind <name> to the function value in the current frame

When a function is called:

1. Add a local frame, titled with the <name> of the function being called
2. Copy the parent of the function to the local frame: [parent= <label>]
3. Bind the <formal parameters> to the arguments in the local frame
4. Execute the body of the function in the environment that starts with the local frame

- An environment is a sequence of frames
- The environment created by calling a top-level function (no def within def) consists of one local frame, followed by the global frame

global
square (x) → func square(x) p=g
triple (x) → func triple (x) p=g
compose (f,g) → func compose1(f,g) p=g
square(5) 25
triple (5) 15
squiple → func compose1(f,g) p=g

f1: square (x) p=g
x 5
rv 25

f2: triple (x) p=g
x 5
rv 15

f3: compose1 (f,g) p=g
f square
g triple
h → func h(x) p=f3
rv

f4: h(x) p=f3
x 5
rv

f5: square (x) p=g
x g(x)
rv

f6: triple(x) p=g
x 5
rv 15

Lamda Expressions

square = lamda x: x * x allows you to evaluate expressions

x → parameter

x * x → actual evaluation part

A function with formal parameter x that returns the value of "x * x"

· no "return" keyword

Environment Diagram

Global Frame

square → func λ(x) (p=g)

λ(x) → doesn't get a name, just a sign

f1: λ (p=g)

4 | + 4
0 | I
13

4+2+6 = 12

9/9/19 Lecture - Iteration

Announcements
- midterm 1 on Monday
- Project Party today
- no recursion on midterm
- not much new content, mostly examples

Return Statements:
- complete evaluation of call expression and provides value
  - f(x): switch to new enviro, execute f's body
  - return statement: switch back to previous enviro
  - only 1 return ever executed

```
def end(n,d):
    while n>0:
        last, n = n%10, n//10
        print(last)
        if d == last:
            return None
```

```
def print_all(k):
    print(k)
    return print_all
```

after running, tells it to print again

print(1)(3)(5)

Gf:
print all → func print_all(k) p=g
f1: print_all [p=g]
k 1
rv → func print_all(k) p=g
f2: print_all [p=g]
k

```
def print_sums(k):
    print(k)
    def next_sum(n):
        return print_sums(n+k)
    return next_sum
```

print sums (1)(3)(5)(7)

print sum (1)

~~print sums (1)(3)~~(5)(7)

1	next_sum (3)	print_sums (1+3=4)
4	next sum (5)	print_sums (4+5=9)
9	next_sum (7)	print_sums (9+7=16)
16		

rv return sum

since no next number, then nothing here

9/11/19 Lecture: Design

Announcements

Functional Abstraction

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum square need to know about square?

- square – one argument — yes
- square – intrinsic name — no
- computes square of number — yes
- computes square w/ mul — no

```
def square(x):
    return pow(x, 2)
```

```
def square(x):
    return mul(x, x-1) + x
```

Choosing Names

- Names don't matter for correctness but matter for composition
- should convey meaning or purpose
- type of value bound best documented in function's docstring
- typically convey their effect, their behavior, value returned

Which Values Deserve a Name

- Repeated compound statements
- Meaningful parts of complex expressions:

More Naming Tips

- Names can be long if they help document your code
- Names can be short if they represent generic quantities

9/12/19 Discussion Notes

Reminders

- go slowly
- don't copy objects on right side
- assign parents
- label frames correctly
- it most likely will not error
- if you have time, check work

Lookup Rule

1. Look for var in current frame

9/13/19 Lecture - Midterm Examples

Function currying → take multi argument into HOF, passing one @ a time

```
def make_adder(n):
    return lambda k: n+k
```

```
def curry2(h):
    def f(x):
        def g(y):
            return h(x,y)
        return g
    return f
```

```
curry2 = lambda h:
            lambda x:
                lambda y:
                    h(x,y)
```

Decorator Function

```
def trace(f):
```

└• @ trace

allows you to follow HOF

WWPD?

- Print returns none, displays its arguments first
-

delayed	delay
delayed	4
6	None

Global Frame

horse └ → func horse(mask) p=g

mask └ → λ(horse) p=g

f1: horse(mask) p=g

mask └ → mask(horse) p=f1

horse └

rv └ 2

* always look to left for names!

f2: λ(horse) p=g

horse └

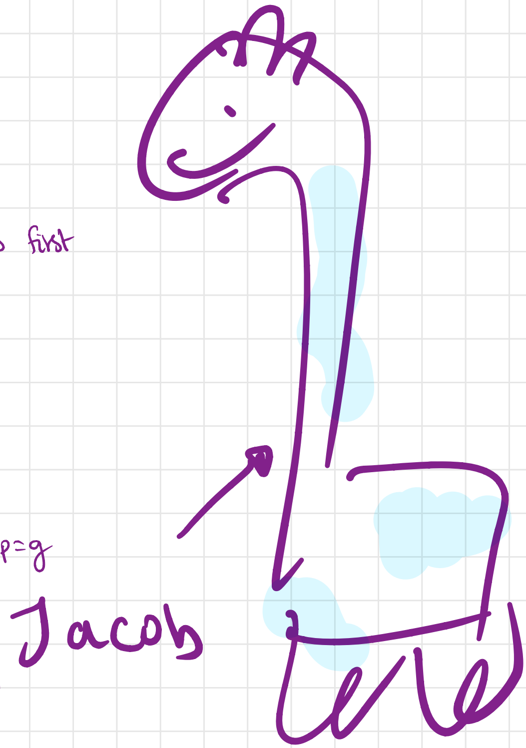
rv └ 2

mask(2)

f3: mask(horse) p=f1

horse └ 2

rv └ 2



9/18/19 - Lecture: Recursion

Recursive: A function is called recursive if the body of that function calls itself, either directly or indirectly

Implication: Executing the body of a recursive function may require applying that function

Digit Sums

$2+0+1+9=12$

- If a number is divisible by 9, then sum_digits(a) is also divisible by 9
- last # of cred card is sum of cred # digits

Sum Dig w/out while:

```
def split(n):
    return n//10, n%10
def sum_digits(n):
    if n<10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

Anatomy of Recursive Function

- def statement header similar to other functions
- conditional statements check for base case

```
def split(n):
    return n//10, n%10
def sum_digits(n):
    if n<10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

Environment Diagrams and Recursive

```
def fact(n):
    if n==0:
        return 1
    else:
        return n * fact(n-1)
fact(3)
```

Global Frame
fact → func fact(n) p=g

F1: fact (p=g)
n 3
rv 6

F2: fact (p=g)
n 2
rv 2

F3: fact (p=g)
n 1
rv 1

F4: fact (p=g)
n 0
rv 1

- fact is called multiple times
- different frame opened for same function

Iteration vs. Recursion

4!

```
def fact_iter(n)
    total, k = 1, 1
    while k ≤ n:
        total, k = total * k, k+1
    return total
```

math: $n! = \prod_{k=1}^{n} k$

names: n, total, k, fact_iter

```
def fact(n):
    if n==0:
        return 1
    else:
        return n * fact(n-1)
```

$$n! \begin{cases} 1 & \text{if } n=0 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

n, fact

Verifying Recursion Functions

The Recursive Leap of Faith:

```
def fact(n):
    if n==0:
        return 1
    else:
        return n * fact(n-1)
```

Is fact implemented correctly?

1. Verify the base case
2. Treat fact as functional abstraction
3. Assume that fact (n-1) is correct
4. Verify that fact(n) is correct

The Luhn Algorithm

```
def luhn_sum_double(n):
    all_but_last, last = split(n)
    luhn_digit = sum_digits
```

Converting Iteration to Recursion

More formulaic: iteration is special case of recursion

idea: the state of an iteration can be passed as arguments

9/18/19 - Discussion: Recursion

1. Base Case (Don't always have to do this first)
2. Break down the problem into smaller recursive calls
3. Use the results of the recursive call to solve problem

   Recursive Leap of Faith!

   Double Check that you've hit the base case

9/20/19 - Lecture: Tree Recursion

Announcements

- regrades due Monday
- HW 3 due thurs 9/26 (v. important)

PRACTICE RECURSION!

Order of Recursive Calls

```
def cascade(n):
    if n<10:
        print n
    else:
        print n
        cascade(n//10)
        print (n)
```

- Each cascade is from a different cascade call
- until return appears, call not completed
- any statement can happen before or after all

- when learning, always put base case first

```
1
1 2
1 2 3
1 2 3 4
1 2 3
1 2
1
```

```
def inverse_cascade(n):
    grow(n)
    print(n)
    shrink(n)
```

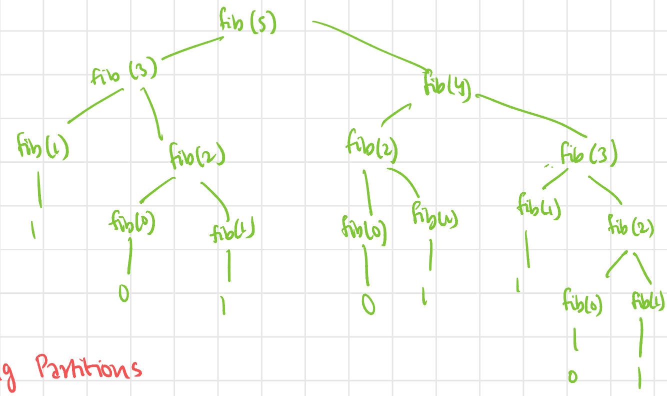
```
grow = lambda n: f_then_g (grow, print, n//10 )
shrink = lambda n: f_then_g (print, shrink, n//10 )
```

```
def f_then_g (f,g,n):
    if n:
        f(n)
        g(n)
```

Tree Recursion

- calls itself more than once in the body
- creates a tree shaped process

```
def fib (n):
    if n==0:
        return 0
    if n==1:
        return 1
    else:
        return fib (n-1) + fib (n-2)
```



Example: Counting Partitions

# of partitions of + int n, using parts to size m, number of ways n can be expressed as sum of positive int parts up to m in inc order

count_partitions(6, 4)

- Recursive decomp: finding similar instances
- Explore 2 possibilities:
  - use atleast one 4
  - don't use any 4
- Solve two simpler problems:
  - count_partitions(2,4)
  - count_partitions(6,3)
- Tree Recursion often involves exploring different choices

negative or zero

```
def count_partitions(n,m):
    if n==0:
        return 1
    elif n<0:
        return 0
    elif m==0:
        return 0
    else:
        with_m = count_partitions(n-m,m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```

9/23/19 - Lecture: Containers

Lists

[41, 43, 47, 49]

odds = [41, 43, 47, 49]

odds[0] = 41

odds[1] = 43

odds[2] = 47

odds[3] = 49

len[odds] = 4

odds[3] - odds[2] = 2

odds[odds[3] - odds[2]] = 47

Working with Limits

digits = [1, 8, 2, 8]

\# of elements:

len(digits) = 4

element selected by its index

digits[3] = 8 getitem(digits, 3) = 8

concatination & repitition

[2, 7] + digits * 2

[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]

add([2, 7], mul(digits, 2))

[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]

nested lists

pairs = [[10, 20], [30, 40]]

pairs[1] = [30, 40]

pairs[1][0] = 30

Containers

digits = [1, 8, 2, 8]

1 in digits

True

'1' in digits

False

[1, 8] in digits

False

[1, [1, 8], 2] = digits → [1, 8] in digits

True

For Statements

Count # times that value is in sequence s

```
count(s, value):
    total, index = 0, 0
    while index < len(s):
        element = s[index]
        if element == value:
```

```
for <name> in <expressions>:
    <suite>
```

```
for element in s:
    element = 1
```

For Statement Execution Procedure

1. Evaluate header <expression>, which must yield an iterable value (a sequence)
2. For each element in that sequence, in order:
   - A. Bind <name> to that element in the current frame
   - B. Execute the <suite>

Range

range(-2,2)

-2, -1, 0, 1 * not 2!

length: ending value − starting value

element selection: starting value + index

list(range(-2, 2))
[-2, -1, 0, 1]

list(range(4))
[0, 1, 2, 3]

* implicitly starts @ 0

makes list of range!

```
for __ in range(3):
    print("Go Bears!")
```

List Comprehension

```
odds = [1, 3, 5, 7, 9]
[x+1 for x in odds]
[2, 4, 6, 8, 10]
```

```
[x for x in odds if 25%x==0]
[1, 5]
[x+1 for x in odds if 25%x==0]
```

Strings

exec('curry...') → does whatever's in string

\n → backslash escapes following character

len(city) → length of string

'here' in 'Where's Waldo?' → True

Dictionaries

num = {'I' : 1, 'V' : 5, 'X' : 10}

>>> {'X' : 10, 'V' : 5, 'I' : 1} → free to shuffle, since they aren't tied down

num['X'] = 10

~~num[10]~~ =

num.keys()

>>> dict_keys(['X', 'V', 'I'])

num.items()

>>> dict_items [

items = [('X', 10), ('V', 5), ('I', 1)]

dict(items)['X']

>>> 10

'X' in numerals

>>> True

numerals.get('X', 0)

>>> 10

{x : x*x for x in range(10)}

>>> {0:0, 1:1, 2:4...}

* can't put lists as keys

Limits on Dictionaries

- Dictionaries are unordered collections of key-value pairs
- Dictionary keys do have two restrictions:
  - A key of a dictionary cannot be a list or a dictionary (or any mutable type)
  - Two keys cannot be equal: There can be at most one value for a given key
- The first restriction is tied to Python's underlying implementation of dictionaries
- If you want to associate multiple values with a key, store them all in a sequence value

9/26/19 – Lecture: Data Abstraction

Data Abstraction

- Compound objects combine objects together
- A date: a year, a month, and a day
- An abstract data type lets us manipulate compound objects as units
- Isolate two parts of any program that uses data:
  - How data are represented (as parts)
  - How data are manipulated (as units)
- Data Abstraction: A methodology by which functions enforce an abstraction barrier between representation and use

Rational Numbers

$\frac{\text{numerator}}{\text{denominator}}$

- Exact representation of integers
- Pair of integers
- As soon as division occurs, exact rep lost!

rational(n, d) → returns a rational number x
numer(x) → returns numerator of x
denom(x) → returns denominator of x

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx * ny}{dx * dy}$$

$$\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx * dy + ny * dx}{dx * dy}$$

```
def mul_rational(x, y)
    return rational(numer(x) * numer(y),
                    denom(x) * denom(y))
```

```
def equal_rational(x, y):
    return numer(x) * denom(y) ==
           numer(y) * denom(x)
```

```
def add_rational(x, y):
    nx, dx = numer(x), denom(x)
    ny, dy = numer(y), denom(y)
    return rational(nx * dy + ny * dy, dx * dy)
```

Pairs

Representing Pairs Using Lists

```
pair = [1, 2]
x, y = pair
x
>>> 1
y
>>> 2
```

A list literal: comma-separated expressions in bracket

"unpacking" a list

```
pair[0]
>>>1
pair[1]
>>>2
getitem(pair, 0)
>>>1
```

Element selection using selection operator

Element Selection function

```
from fractions import gcd
g = gcd(n, d)
return [n//g, d//g]
```

Abstraction Barrier

Parts of program that...	treat rational as...	using...
use rational numbers to perform computation	whole data values	add_rational, mul_rational, rationals_are_equal, print_rational
create rationals or implement rational operators	numerator and denominator	rational, numer, denom
implement selectors and constructors for rationals	two-element lists	list literals and element selection

implementation of lists

Violating Abstraction Barrier

```
add_rational([1, 2], [1, 4])
```

no constructor!

should be rational (1, 2), rational (1, 4)

```
def divide_rational(x, y):
    return [x[0] * y[1], x[1] * y[0]]
```

wrong!

rational(numer(x) denom(y) denom(x) numer(y))

Data Representation

What is Data?

- We need to guarantee that constructor and selector functions work together to specify the right behavior
- Behavior condition: If we construct rational number x from numerator n and denominator d, then numer(x)/denom(x) must equal n/d
- Data abstraction uses selectors and constructors to define behavior
- If behavior conditions are met, then the representation is valid

You can recognize data abstraction by its behavior!

9/25/19 - Lecture: Trees

Box and Pointer Notation

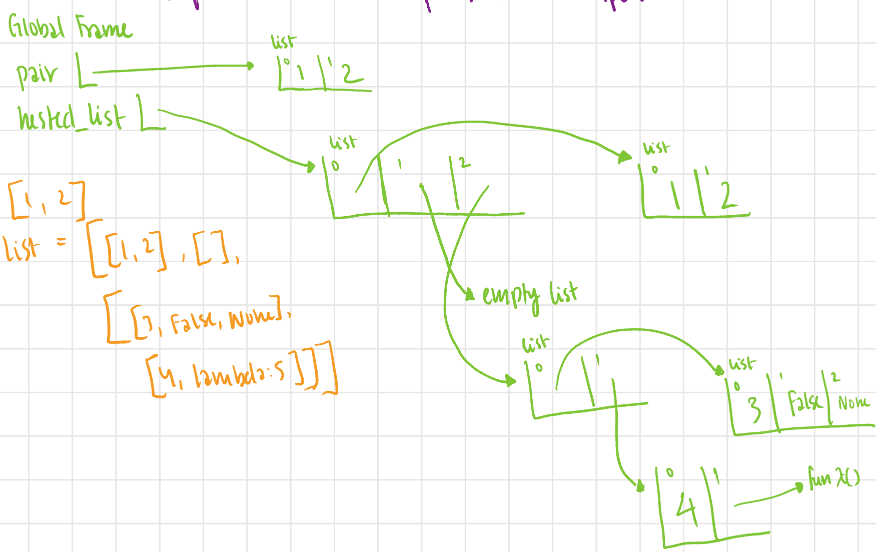
The Closure Property of Data Types

- A method for combining data values satisfies the closure property if:
  The result of combination can itself be combined using the same method
- Closure is powerful because it permits us to create hierarchical structures
- Hierarchical structures are made up of parts, which themselves are made up of parts, and so on

Lists can contain lists as elements (in addition)

Box-and-Pointer Notation in Environment Diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element
Each box either contains a primitive value or points to a compound value




```
pair = [1, 2]
nested_list = [[1, 2], [],
    [[3, False, None],
     [4, lambda: 5]]]
```

Slices

```
odds = [3, 5, 7, 9, 11]
list(range(1, 3))
  [1, 2]
[odds[i] for i in range(1, 3)]
  odds[1:3]      odds[:3]      odds[1:]
    [5, 7]       [3, 5, 7]     [5, 7, 9, 11]
```

* no first value : beginning
* " last " : end

Slicing Creates New Values

every time you slice, it creates a copy of the list, not changing the actual value

Processing Container Values

Sequence Aggregation

Several built-in functions take iterable arguments and aggregate them into a value

sum(iterable [, start]) → value

Return the sum of an iterable of numbers (NOT strings) plus the value of parameter 'start' (which defaults to 0). When the iterable is empty, return start

sum([2,3,4]) = 9 sum([2,3,4], 5) = (9+5) = 14

[2,3] + [4] = [2,3,4]

max(iterable [, key=func]) → value

max(a,b,c ... [, key=func]) → value

With a single iterable argument, return its largest item

With two or more arguments, return the largest argument

* also min & any

max(range(10), key=lambda x: $\underbrace{7-(x-4)*(x-2)}$)

max of those values that are outputs of this function

all(iterable) → bool

Return True if bool(x) is True for all values x in the iterable

If the iterable is empty, return True

bool(True) bool(Hello)

True True

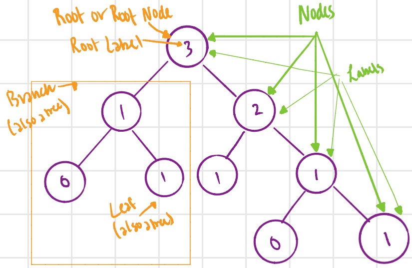
all([x<5 for x in range(5)])

True

all(range(5)) 0 1 2 3 4

False * 0 is false

Trees



Recursive description (wooden trees):

A tree has a root label and a list of branches

Each branch is a tree

A tree with zero branches is called a leaf.

Relative description (family trees):

Each location in a tree is called a node

Each node has a label that can be any value

One node can be the parent/child of another

* People often refer to labels by their locations:

"each parent is the sum of its children"

Implementing the Tree Abstraction

```
tree (3, [tree(1),
          tree (2,[tree(1),
                   tree(1)])])
[3, [1], [2, [1], [1]]]
```

- A tree has a root label and a list of branches
- Each branch is a tree

```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)
```

```
def is_tree(tree):
    if type(tree) != list or len(tree)<1:    * returns
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True
```

```
def label(tree):     * returns top value of tree
    return tree[0]
```

```
def branches(tree):     * returns list disculding 1st value
    return tree [1:]
```

```
def is_leaf (tree):     * returns True if branch is a leaf
    return not branches (tree)
```

Tree Processing Uses Recursion

Processing a leaf is often the base case of a tree processing function
The recursive case typically makes a recursive call on each branch, then aggregates

```
def count_leaves(t):
    if is_leaf(t):
        return 1
    else:
        branch_counts = [count_leaves(b) for b in branches(t)]
        return sum(branch_counts)
```

Discussion Question

Implement leaves, which returns a list of the leaf labels of a tree
Hint: If you sum a list of lists, you get a list containing the elements of those lists

```
sum([[1], [2,3], [4]], [])
[1,2,3,4]
```

```
def leaves(tree):
    if is_leaf(tree):
        return [label(tree)]
    else:
        return sum(List of leaves for each branch, [])
```

Creating Trees

A function that creates a tree from another tree is typically also recursive

```
def increment_leaves(t):
    if is_leaf(t):
        return tree(label(t)+1)
    else:
        bs = [increment_leaves(b) for b in branches(t)]
        return tree(label(t), bs)
```

```
def increment(t):
    return tree(label(t)+1, [increment(b) for b in branches(t)])
```

9/30/19 - Lecture: Mutable Values

Objects

today.strftime('%A %B %d')
'Monday, September 30'

month

day (#)

day of week

today.year
2019

today
datetime.date(2019, 9, 30)

DeNero :)

- objects rep info
- consist of data & behavior, bundled to create abstraction
- can rep things, also properties, interactions, processes
- type of object: class, classes are first-class in Python
- Object-Oriented Programming
  - metaphor for organizing large programs
  - special syntax to improve code
- In Python, every value is object
  - all objects have attributes
  - data manipulation happens through object methods
  - functions do one thing, objects do many related things

Examples: Strings

'Hello'.<some function>()

ASCII code chart! String <-> integers

suits = ['coin', 'string', 'myriad']

original = suits

suits.pop()

'myriad'

suits.remove('string')

suits.append('cup')

suits.extend(['sword', 'club'])

list pop: pops off last value of list

suits
['coin', 'cup', 'sword', 'club']

suits
['coin', 'cup', 'spade', 'club']

suits
['heart', 'diamond', 'spade', 'clubs']

original
"

* object was being changed, therefore both vars will show the same

Some Objects Can Change

^^ object changing state

Same object can change through functions

* only dictionaries and lists can be mutated

all

```
numerals.pop('V')
    5
numerals
{'I':1, 'X':10}
```

* dictionary pop: pops off key entered

Mutation can Happen Within Function Call

A function can change the value of any object in its scope

```
four = [1,2,3,4]
 len(four)
  4
 mystery(four)
   len(four)
  2
```

```
def mystery(s)
  s.pop()
  s.pop()
```

```
def mystery(s)
  s[2:] = []
```

Tuples

[2,3,4,5]	(2,3,4,5)	2,3,4,5	()
list	tuple	tuple	empty tuple

```
t = (2,3,4,5)
t[0] = 2
  t[:3] = (2,3,4)
```

```
tuple()
 ()
```

(2,)
└ 1 element touple

* can switch between list & tuple

tuple([2])

(3,4) + (5,6)
(3,4,5,6)

(3,4)*2
(3,4,3,4)

* multiplying makes a new copy of tuple, not changing original list

t = (2,3,4,5)
t[2] = 7

* cannot change contents of a tuple

tuples can be keys in dict

Tuples are Immutable Sequences

- Immutable values are protected from mutation
- Value of expression can change bc changes in names or objects
- may still change if it contains a mutable value as an element

s = ([1,2],3)   s= ([1,2],3)

~~s[0] = 4~~   s[0][0] = 4 ✓

Sameness and Change

- as long as we never modify objects, a compound object is just the totality of its pieces
- a rational number is just its numerator and denominator
- this view is no longer valid in the presence of change
- a compound data object has an "identity" in addition to the pieces of which it is composed
- a list is still "the same" list even if we changed its contents
- Converseley, we could have two lists that happen to have the same contents, but are different

Identity Operators

Identity

<exp0> is <exp1>

evaluates to True if both evaluate to same object

Equality

<expo> == <exp1>

evaluates to True if both <expo> and <exp1> evaluate to equal values

Identical Objects are always equal values

10/02/19 - Lecture: Mutable Functions

* change values of variable in the parent frame in a smaller function w/ non local variables

- must have already been used above
- can't use nonlocal var after already in local frame

* local & nonlocal lookup of balance produces error!
* doesn't do that to list since list is mutable
- mutable function - variable in function always changing
- john and steven not equal, since calls different functions w/ different funds
- even if same john and steven amount, still not equal

- referentially transparent ≠ mutable functions
  making it 10 + b(4) different b(3) + b(4)
  since x isn't changed

A Function With Behavior that Varies Over Time

Model a Bank Account w/ balance of $100

withdraw(25) ← argument - amount to withdraw

returning value: remaining balance → 75

withdraw(25) ← 2nd withdrawl of same amount

50

different return value! → withdraw(60

insufficient funds

* use higher-order function:

withdraw = make_withdraw(100)

How is balance stored?
- within parent frame of function
- function has body & parent environment

Reminder: Local Assignment

```
def percent_difference(x,y):
    difference = abs(x-y)
    return 100 * difference/x
diff = percent_difference(40,50)
```

→ Assignment binds name(s) to value(s) in the first frame of current environment

Execution Rule for Assignment Statements:

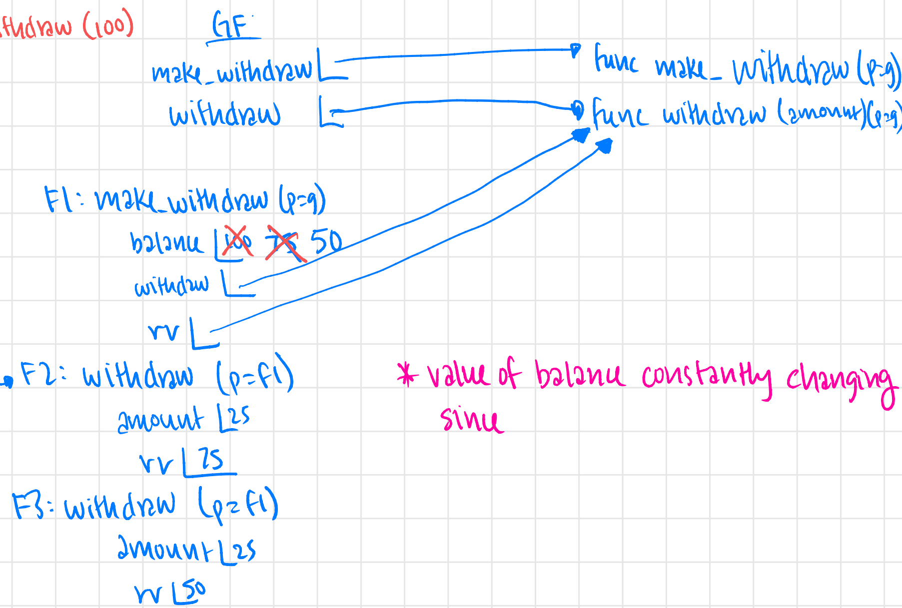
1. Evaluate all expressions right of =, from left to right
2. Bind the names on the left to the resulting values in the current frame

Non-Local Assignment & Persistent Local State

```
def make_withdraw(balance):
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return 'Insufficient funds'
        balance = balance-amount
        return balance
    return withdraw
withdraw = make_withdraw(100)
withdraw(25)
withdraw(25)
```

nonlocal balance → declare the name "balance" nonlocal at the top of the body of the function in which it is re-assigned

balance = balance-amount → rebind balance in 1st non-local frame in which it was bound previously



* value of balance constantly changing since

The Effects of Nonlocal Statements

nonlocal <name>

Effect: Future assignments to that name change its pre-existing binding in the first non local frame of the current environment in which that name is bound

- non-local variable must be referenced to before function
- cannot have same variable in existing frame

The Many Meanings of Assignment Statements

Status	Effect
• no nonlocal statement "x" not locally bound	• create new binding from name "x" to 2 in first frame of current environment
• no nonlocal statement "x" is bounded locally	• rebound "x" to 2 in first frame of current environment
• nonlocal x "x" is bound in non-local frame	• rebind x to 2 in first non-local frame of the current environment in which it was bound
• nonlocal x x is not bound in a non-local frame	• SyntaxError: no binding for nonlocal 'x' found
• nonlocal x x is bound in nonlocal frame x also bound locally	• Syntax Error: name 'x' is parameter and nonlocal

Python Particulars

Python pre-computes which frame contains each name before executing the body of a function

Within the body of a function, all instances of a name must refer to the same frame.

```
def make_withdraw(balance):
    def withdraw(amount):
        if amount > balance:
            return 'Insufficient funds
        balance = balance - amount
        return balance
    return withdraw

wd = make_withdraw(20)
wd(5)
```

balance created in separate frame

local assignment but

Mutable Values

- mutable values can be changed without nonlocal statement

```
def make_withdraw_list(balance):
    b = [balance]
    def withdraw(amount):
        if amount > b[0]:
            return 'Insufficient funds'
        b[0] = b[0] - amount
        return b[0]
    return withdraw

withdraw = make_withdraw_list(100)
withdraw(25)
```

name bound outside of withdraw def

element assignment changes a list

Multiple Mutable Functions

john = make_withdraw(100)
steven = make_withdraw(100000)

john is not steven
True

John == steven
False

* even if both have same amount in bank, still won't be equal, only john(0) == steven(0) will be equal

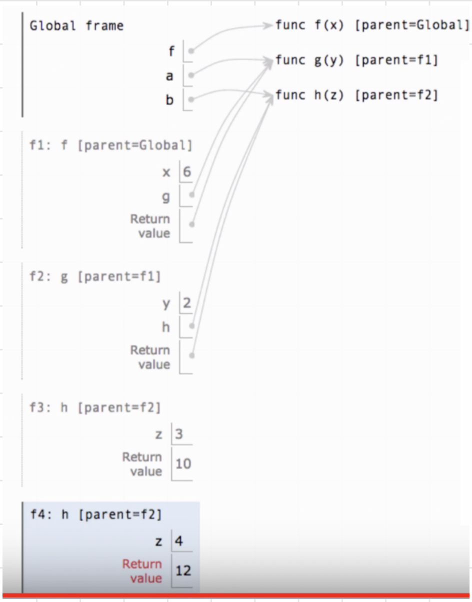
Referential Transparency Lost

- Expressions are referentially transparent if substituting an expression with its value does not change the meaning of a programe

  mul(add(2, mul(4,6)), add(3,5))
  mul(26, add(3,5))

- Mutation operations violate the condition of referential transparency because they do more than just return a value; they change the environment

· referentially transparent ≠ mutable functions
making it 10 + b(4) different b(3) + b(4)
since x isn't changed

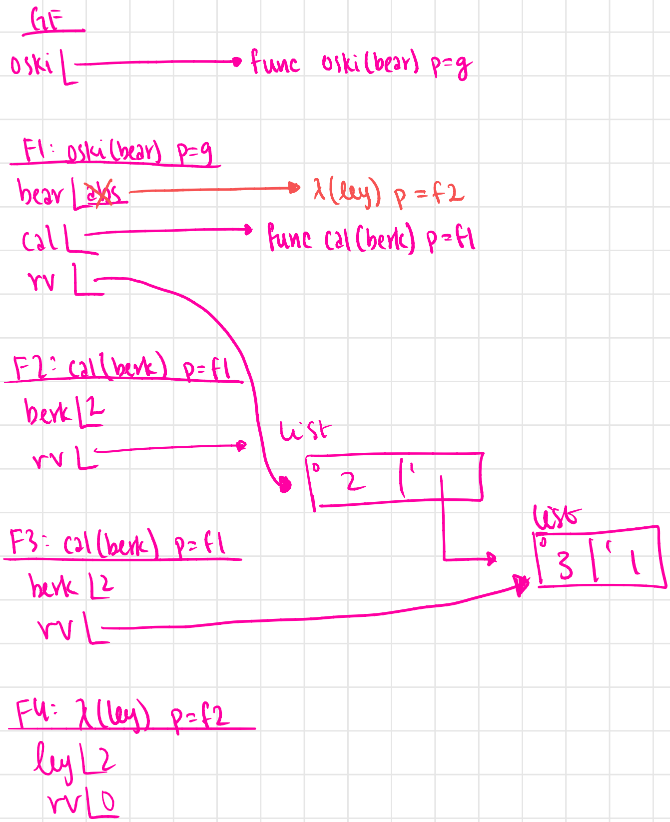



```
def f(x):
    x = 4
    def g(y):
        def h(z):
            nonlocal x
            x = x + 1
            return x + y + z
        return h
    return g
a = f(1)
b = a(2)
total = b(3) + b(4)
```

Environment Diagrams

Go Bears!

```
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)
oski(abs)
```



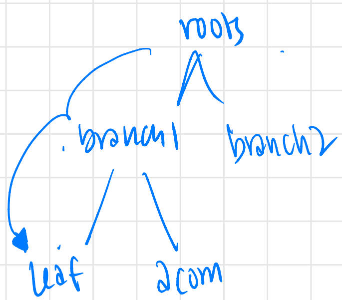
10/03/19 : Labs - Data Abstraction, Trees

Data Abstraction

- allows you to treat any code as an object
- constructor: functions that build the abstract data type
- selectors: functions that retrieve information from the data type

Trees

- tree - data structure that represents hierarchy of information
- Constructor: tree(label, branches = []):
- Selectors: label(tree) → return value in root node of tree
  branch(tree) → return list of branches in given tree
  is_leaf(tree) → returns True if tree's list of branches is empty and False otherwise



iteralos

iterators

lecture

10/05/19: Lecture: Iterators

Iterators

A container can provide an iterator that provides access to its elements in some order

iter(iterable): return an iterator over the elements of an iterable value

next(iterator): return the next element in an iterator

s = [3, 4, 5]

t = iter(s) u = iter(s)

next(t) → 3 next(u) → 3 next(u) → 4

next(t) → 4 next(t) → 5

* iter creates the iterator, next calls the next value to iterate on

* new iterators over same value doesn't mess with old one

s = [[1, 2], 3, 4, 5]

t = iter(s)

next(t) → [1, 2]

next(t) → 3

list(t) → [4, 5] * displays what's left

next(t) → StopIteration * error showing end of iteration

Dictionary

Views of a Dictionary

- An iterable value is anything that can be passed to iter to produce an iterator
- An iterator is returned from iter and can be passed to next; all iterators are mutable
- A dictionary, its keys, its values, and its items are all iterable values
  - The order of items in a dictionary is the order in which they were added
  - Historically, items appeared in an arbitrary order

d = {'one': 1, 'two': 2, 'three': 3}

d['zero'] = 0

k = iter(d.keys()) (or iter(d))	v = iter(d.values())	i = iter(d.items())
next(k) → 'one'	next(v) → 1	next(i) → ('one', 1)
next(k) → 'two'	next(v) → 2	next(i) → ('two', 2)
next(k) → 'three'	next(v) → 3	next(i) → ('three', 3)
next(k) → 'zero'	next(v) → 0	next(i) → ('zero', 0)
(d.keys()) → just key	(d.values()) → just value	(d.items()) → tupple of both

d = {'one': 1, 'two': 2}
k = iter(d)
d['zero'] = 0
next(k) → produces an error since size of dictionary changed

d = {'one': 1, 'two': 2, 'zero': 1}
k = iter(d)
d = ['zero'] = 0
wouldn't error

* can modify already created keys within dictionary, but cannot add new ones

For Statements

r = range(3, 5)
list(r) →
[3, 4, 5]

ri = iter(r)
next(ri) → 3
for i in ri
    print(i) → 4, 5

for loop can work w/ iterable function, immediately goes through all items to end

Built-in Functions for Iteration

Many built-in Python sequence operations that return iterators that compute results lazily
- map (func, iterable): Iterate over func(x) for x in iterable
- filter (func, iterable): Iterate over x in iterable if func(x)
- zip (first_iter, second_iter): Iterate over co-indexed (x, y) pairs
- reversed (sequence): Iterate over x in a sequence in reverse order

To view the contents of an iterator, place the resulting elements into a container
- list (iterable): Create a list containing all x in iterable
- tuple (iterable): create a tuple containing all x in iterable
- sorted (iterable): create a sorted list containing x in iterable

bcd = ['b', 'c', 'd']
[x.upper() for x in bcd] → ['B', 'C', 'D']
m = map(lambda x: x.upper(), bcd) * if just map(blah, blah) then it would just return a function
next(m) → 'B'
next(m) → 'C'
next(m) → 'D'
next(m) → StopIteration

m = map(double, range(3,7))
f = lambda y: y >= 10
t = filter(f, m)

next(t) → 10
3 => 6 **
4 => 8 **
5 => 10 **

list(filter(f, map(double, range(3,7))))
[10, 12]