There's a poll: pollev.com/cs61a

Midterm Review (Part 2)

Announcements

Describing Functions

Describing Higher-Order Functions

```
def exp(x):
    def base(b):
        return pow(b, x)
    return base
```

```
def exp(x):
    return lambda b: pow(b, x)
```

The exp function takes an exponent **x** and then a base **b** and raises the base to the exponent:

```
exp(2)(3) == 9

square = exp(2)  # the square function takes a base b and squares it.
```

Environments ensure that the function exp(2) will behave the same way anywhere it's called.

(Demo)

Example: Reverse

The square function can be defined in terms of the built-in pow function:

```
def square(x):
    """Square x.

    >>> square(3)
    9
    """
    return pow(x, 2)
```

```
def cube(x):
    """Cube x.

    >>> cube(3)
    27
    """
    return pow(x, 3)
```

Define square and cube in one line without using lambda or ** (using **curry** and **reverse** and **pow**).

```
def reverse(f):
    return lambda x, y: f(y, x)
```

Reverse switches the argument order of a two-argument function
reverse(pow)(2, 3) == pow(3, 2)

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

Curry converts a two-argument function into a function that takes the first and then the second argument in two calls:
curry(pow)(5)(2) == pow(5, 2)

```
square = curry(reverse(pow))(2)
cube   = curry(reverse(pow))(3)
```

Poll for this one is on pollev.com/cs61a

Spring 2025 Question 1

```
def double(x):
    return 2 * x
def square(f):
    return lambda x: f(x) * f(x)
def inc(f):
    return lambda x: f(x + 1)
def triple(f):
    return lambda x: f(f(f(x)))
def put(x):
    return lambda f: f(x)
```

Doubles a number

Takes f then x and then squares f(x)

Takes f then x and then calls f on x+1

Takes f then x and calls f 3 times on x

Takes x then f and then calls f(x)

✅ `triple(print)(5)` Calls print 3 times on 5

```
print(print(print(5)))    5
                 None     None
          None            None
```

`one = put(1)` Takes f and calls f(1)

✅ `one(square(double))` square(double) squares the double of x, so square the double of 1 — 4

✅ `inc(put)(1)(triple(double))`

```
triple(double) doubles 3 times: (2 * (2 * (2 * x)))
inc(put)(1) calls put on 1+1:   Takes f and calls f(2)
So we call triple(double) on 2: (2 * (2 * (2 * 2)))
```

16

pollev.com/cs61a

https://pythontutor.com/cp/composingprograms.html#code=def%20double%28x%29%3A%0A%20%20%20%20return%202%20*%20x%0A%0Adef%20square%28f%29%3A%0A%20%20%20%20return%20lambda%20x%3A%20f%28x%29%20*%20f%28x%29%0A%0Adef%20inc%28f%29%3A%0A%20%20%20%20return%20lambda%20x%3A%20f%28x%20%2B%201%29%0A%0Adef%20triple%28f%29%3A%0A%20%20%20%20return%20lambda%20x%3A%20f%28f%28f%28x%29%29%29%0A%0Adef%20put%28x%29%3A%0A%20%20%20%20return%20lambda%20f%3A%20f%28x%29%0A%20%20%20%20%0Aone%20%3D%20put%281%29%0Aprint%28one%28square%28double%29%29%29%0A&cumulative=true&curInstr=25&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

Substitution

Substitution and Environments

Environment diagrams show how Python actually calls functions

Substition is another (simpler) way to describe calling functions (with no assignment)

Calling a function...

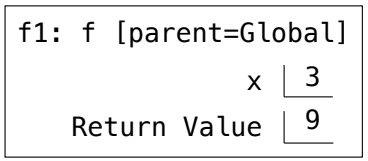
If you see an assignment statement, don't try to use substition!

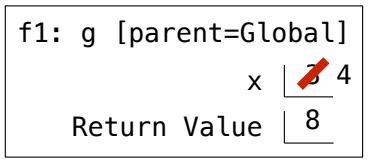
```
def f(x):
    return x * x
f(3)
```

```
def g(x):
    x = x + 1
    return 2 * x
g(3)
```

Using **Environments:**

1. Add a local frame
2. Bind parameters to arguments
3. Execute the function's body





Using **Substitution:**

1. Replace the parameters with arguments
2. Execute the function's body

```
return 3 * 3
```

```
3 = 3 + 1
return 2 * 3
```

Spring 2025 Question 1 Using Substitution

```
def double(x):
    return 2 * x
def square(f):
    return lambda x: f(x) * f(x)
def inc(f):
    return lambda x: f(x + 1)
def triple(f):
    return lambda x: f(f(f(x)))
def put(x):
    return lambda f: f(x)
```

```
lambda x: double(x) * double(x)
```

square(double) squares the double of x

```
lambda f: f(1)
```

Takes f and calls f(1)

```
one = put(1)
```

```
one(square(double))
```

```
one(lambda x: double(x) * double(x))
   (lambda x: double(x) * double(x))(1)
              double(1) * double(1)
                2 * 1   *   2 * 1
```

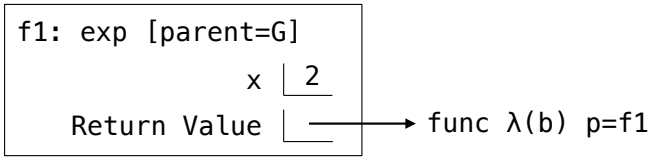
Substitutions and Inner Functions

The exp function takes an exponent **x** and then a base **b** and raises the base to the exponent:

```
def exp(x):
    return lambda b: pow(b, x)
```

```
lambda b: pow(b, 2)

square = exp(2)
```



Environments & Programs

Python Uses Static Scope

Scope of a name: the parts of a program that can refer to that name

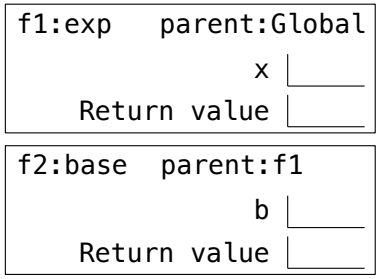
Static scope means you can refer to local names, enclosing function names, and global names

For any **expression**, you can know the function names of the frames it will be **evaluated in** *without drawing the whole environment diagram*

- For any expression that appears outside a **def** statement or **lambda** expression, the environment is just the global frame
- For other expressions: the environment has a frame for every enclosing **def** statement or **lambda** expression ordered from inner to outer

```
def exp(x):
    def base(b):
        return pow(b, x)
    return base
```

Will always be evaluated in an environment with 3 frames: **base**, **exp**, and then **global**



That means, for any **name**, you can know the function name of the frame it will be **found in** *without drawing the whole environment diagram*

Checking Your Work: Spring 2025 Question 2

```
 1: x, y = 1, 2
 2:
 3: def this(x):
 4:     x = 3
 5:     print(y)
 6:     return 4
 7:
 8: def which(x, f):
 9:     def this(x):
10:         x = 5
11:         return y
12:     y = 6
13:     y = f(x)
14:     print(this(y))
15:     return lambda: (print(x) or y)
16:
17: print(which(7, this)())
```

frames: this, global ; the global frame contains **y**

frames: this, which, global ; the which frame contains **y**

frames: which, global ; the which frame contains **this** & **y**

pollev.com/cs61a

frames: lambda, which, global
the which frame contains **x** & **y**

frames: global ; the global frame contains **which** and **this**

https://pythontutor.com/cp/composingprograms.html#code=x,%20y%20%3D%201,%202%0A%0Adef%20this%28x%29%3A%0A%20%20%20%20x%20%3D%203%0A%20%20%20%20print%28y%29%0A%20%20%20%20return%204%0A%0Adef%20which%28x,%20f%29%3A%0A%20%20%20%20def%20this%28x%29%3A%0A%20%20%20%20%20%20%20%20x%20%3D%205%0A%20%20%20%20%20%20%20%20return%20y%0A%20%20%20%20y%20%3D%206%0A%20%20%20%20y%20%3D%20f%28x%29%0A%20%20%20%20print%28this%28y%29%29%0A%20%20%20%20return%20lambda%3A%20%28print%28x%29%20or%20y%29%0A%0Aprint%28which%287,%20this%29%28%29%29%0A&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

Functional Abstraction Practice

Why Are So Many Questions About Digits?

So far we've learned about:

- **Arithmetic:** +, -, *, /, //, %, ==, <, <=, >, >=, !=, pow, max, min, abs
- **Logic:** and, or, not
- **Assignment:** =
- **Control:** def, lambda, (), if, elif, else, while

Computer Science is the study of what can be done with these elements

The same way we work with digits allows us to work with more interesting data (next week)

Spring 2025 Question 3(a)

The function **all_digits** takes a positive integer **n** and one-argument function **cond** that always returns **True** or **False.** It returns **True** if **cond(d)** returns **True** when called on every digit **d** in **n,** and **False** otherwise**.**

```
def all_digits(n, cond):
    """Return whether cond returns true for every digit of positive n.

    >> odd = lambda d: d % 2 == 1
    >>> all_digits(123, odd)  # not all digits are odd
    False
    >>> all_digits(357, odd)  # all digits are odd
    True
    """
    while n:
        if ____not cond(n % 10)____:
            ____return False____
        n = n // 10
    return ____True____
```

From discussion:

Describe a process (in English) that computes the output from the input using simple steps.

Questions to help figure out the process:

- Can you ever stop before going through all of the digits?
- What kind of value should you return?

Spring 2025 Question 3(b)

From 3(a): the function **all_digits** takes a positive integer **n** and one-argument function **cond** that always returns **True** or **False.** It returns **True** if **cond(d)** returns **True** when called on every digit **d** in **n,** and **False** otherwise.

Definition. A prefix of a positive integer **n** is the value of **n//pow(10, p)** for some non-negative integer **p.** For example, 3456 has prefixes 3456, 345, 34, 3, and 0.

```
def prefix_digits(n, cond):
    """Return the largest prefix of positive n for which cond returns true for every digit.

    >>> odd = lambda d: d % 2 == 1
    >>> prefix_digits(94720, odd)
    9
    >>> prefix_digits(919321, odd)
    9193
    >>> prefix_digits(2025, odd)
    0
    >>> prefix_digits(20252025, lambda d: d < 4)
    202
    >>> prefix_digits(20252025, lambda d: True)
    20252025
    """
    return process(n, lambda k: _____ )
```

```
def process(n, check):
    """A function to help implement prefix_digits."""
    while n:
        if check(n):
            return n
        n = n // 10
    return 0
```

Important questions:

- What does **process** do?
- How does that help with **prefix_digits**?
- What does the **k** represent in **lambda k:____**?

all_digits(k, cond)

Iteration Practice

Spring 2025 Question 3(c)

Definition. A prefix of a positive integer **n** is the value of **n//pow(10, p)** for some non-negative integer **p.** For example, 3456 has prefixes 3456, 345, 34, 3, and 0.

```
def prefix_digits(n, cond):
    """Return the largest prefix of positive n for which cond returns true for every digit.
    >>> odd = lambda d: d % 2 == 1
    >>> prefix_digits(94720, odd)
    9
    >>> prefix_digits(919321, odd)
    9193
    >>> prefix_digits(2025, odd)
    0
    >>> prefix_digits(20252025, lambda d: d < 4)
    202
    """
    k = 0
    while n >= _____pow(10, k)_____ :
        if cond(_n // pow(10, k) % 10_):
            k += 1
        else:
            n = n // 10
            _________k = 0_________
    return n
```

Questions to help figure out the process:

- Can you ever stop before going through all of the digits?
- What kind of value should you return?
- What does k represent?
- How does an increasing k help?

Process that fits this template:

- Use k to call cond on each digit of n
- If cond ever returns false, shorten n and start over

Spring 2025 Question 4

```
def hailstone(n):
    """Print numbered updates in the hailstone sequence.

    >>> hailstone(10)
    1 10 -> 5
    2 5 -> 16
    3 16 -> 8
    4 8 -> 4
    5 4 -> 2
    6 2 -> 1
    """
    def f():
        if n % 2 == 1:
            m = 3 * n + 1
        else:
            m = n // 2
        print(k, n, '->', m)
        return m
    k = 1
    while n > 1:
        k, n = k + 1, f()
```

Questions to help figure out the process:

- What does **f** do?
- What do **m**, **n**, and **k** represent?