

Announcements From Others

CodeBase

"CodeBase is a student-led consultancy that works with local startups to build applications, future product iterations, and develop algorithms. This semester, we're working with three startups to create a cross-platform mobile application, develop an Artificial Intelligence Chatbot, and build data integrations for internet-connected devices like Amazon Alexa. You can find out more about us at codebase.berkeley.edu."

Engineering Solutions at Berkeley

"Are you an engineering or computer science student interested in consulting or internship-style projects? If so, apply now to Engineering Solutions at Berkeley (ES)! ES is a new, pro-bono consulting club unlike any on campus. We use our technical expertise to solve engineering challenges our corporate partners contract to us during the school year. This year we are creating an automated progress reporting system for a multinational construction company. One aspect of the project is using machine learning to perform predictive analysis on how likely a project is to be completed on-time and on-budget in real-time. If this sounds of any interest to anyone, come to our info-session next Thursday, January 26th from 7-8 pm in 228 Dwinelle!!! RSVP for the info-session here: <https://goo.gl/forms/YH3JVAHTJUm4dG1F9> Apply by 11:59 pm on Sunday, January 29th here: <https://esberkeley.com/join/>"

Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 1

Official Announcements

- Test #1 is Friday, 17 February, 7-9PM. Rooms to be announced.

Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 2

Lecture #4: Control (cont.) and Higher-Order Functions

Explicit Repetition

- But in the Python, C, Java, and Fortran communities, it is more usual to be explicit about the repetition.
- The simplest form is **while**:

```
while Condition:
    Statements

def loop():
    if Condition:
        Statements
    loop()
```

means "If condition evaluates to a true value, execute statements and repeat the entire process. Otherwise, do nothing."

- The effect is (nearly) identical to

```
def loop():
    # Start things off
    loop()
```

- ... **except** that (for most Python implementations) the latter eventually runs out of memory; **and** we'll have to do something about assignments to variables in **Statements** (more on that later).

Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 5

Indefinite Repetition

- With conditionals and function calls, we can conduct computations of any length.
- For example, to sum the squares of all numbers from 1 to N (a parameter):

```
def sum_squares(N):
    """The sum of K**2 for K from 1 to N (inclusive)."""
    if N < 1:
        return 0
    else:
        return N**2 + sum_squares(N - 1)

sum_squares(3) => 3**2 + sum_squares(2)
=> 3**2 + 2**2 + sum_squares(1)
=> 3**2 + 2**2 + 1**2 + sum_squares(0)
=> 3**2 + 2**2 + 1**2 + 0 => 14
```

- This will repeatedly call `sum_squares` with decreasing values (down to 1), adding in squares: Execute here

Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 4

Sum_squares Iteratively?

- Our original `sum_squares` was

```
def sum_squares(N):
    """The sum of K**2 for K from 1 to N (inclusive)."""
    if N < 1:
        return 0
    else:
        return N**2 + sum_squares(N - 1)
```

- How do we do the same thing with a **while** loop?

```
def sum_squares(N):
    """The sum of K**2 for K from 1 to N (inclusive)."""
```

Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 6

Sum_squares Iteratively (II)

```
def sum_squares(N):
    """The sum of K**2 for K from 1 to N (inclusive)."""
    result = 0
    k = 1
    while k <= N:
        result += k**2
        k += 1
    return result
```

Execute this

Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 7

Another Way

- Alternatively, I can make this a little shorter by adding the other way:

```
def sum_squares(N):
    """The sum of K**2 for K from 1 to N (inclusive)."""
    result = 0
    while N >= 1:
        result += N**2
        N -= 1
    # Or result = result + N**2
    # Or N = N-1
    return result
```

Execute here

Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 8

Functions As Templates

- If we think of a function body as a template for a computation, parameters are “blanks” in that template.

- For example:

```
def sum_squares(N):
    k, sum = 0, 0
    while k <= N:
        sum, k = sum+k**2, k+1
    return sum
```

is a template for an infinite set of computations that add squares of numbers up to 0, 1, 2, 3, ..., in place of the **N**.

Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 9

Functions on Functions

- Likewise, function parameters allow us to have templates with slots for *computations*:

```
def summation(N, f):
    k, sum = 1, 0
    while k <= N:
        sum, k = sum+f(k), k+1
    return sum
```

- Generalizes `sum_squares`: We can write `sum_squares(5)` as:

```
def square(x):
    return x**x
summation(5, square)
```

- or (if we don't really need a “square” function elsewhere) we can create the function argument anonymously on the fly:

```
summation(5, lambda x: x**x)
```

Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 10

Lambda

- In Python, `lambda` is just an abbreviation.

- Writing `lambda PARAMS: EXPRESSION` is the same as writing `NAME`, where `NAME` is a name that appears nowhere else in the program and is defined by

```
def NAME(PARAMS):
    return EXPRESSION
```

evaluated in the same environment in which the original `lambda` was.

- Now we can write any number of summations succinctly:

```
summation(10, lambda x: x**3)      # Sum of cubes
summation(10, lambda x: 1 / x)    # Harmonic series
summation(10, lambda k: x**(k-1) / factorial(k-1))
# Approximate e**x
```

Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 11

Functions that Produce Functions

- Functions are *first-class values*, meaning that we can assign them to variables, pass them to functions, and return them from functions.

- Example: let's generalize the class of functions that—like

```
def h(x): return sin(x) + cos(x)
```

—add the results of applying two functions to the same argument:

```
>>> def add_func(f, g):
...     """Return function that returns F(x)+G(x) for argument x."""
...     def adder(x):
...         return f(x) + g(x)
...         # or return lambda x: f(x) + g(x)
...     return adder
>>> from math import sin, cos, pi
>>> h = add_func(sin, cos)
>>> sin(pi/4) + cos(pi/4)
1.414213562373095
>>> h(pi / 4)
1.414213562373095
```

Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 12

Generalize!

- Let's make a general function-combining function (that goes beyond addition):

```
>>> def combine_funcs(op):
...     """combine_funcs(OP)(f, g)(x) = OP(f(x), g(x))."""
...     def combined(f, g):
...         def val(x):
...             return op(f(x), g(x))
...         return val
...     return combined
```

- Now `add_func` itself can be constructed by a call to `combine_funcs`:

```
>>> from operator import add
>>> add_func = combine_funcs(add)
>>> from math import sin, cos, pi
>>> h = add_func(sin, cos)
>>> h(pi / 4)
1.414213562373095
```

- What do the environments look like here? Think about it and try it out.

Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 13

Generalize!

- Let's make a general function-combining function (that goes beyond addition):

```
>>> def combine_funcs(op):
...     """combine_funcs(OP)(f, g)(x) = OP(f(x), g(x))."""
...     def combined(f, g):
...         def val(x):
...             return op(f(x), g(x))
...         return val
...     return combined
```

- Now `add_func` itself can be constructed by a call to `combine_funcs`:

```
>>> from operator import add
>>> add_func = combine_funcs(add)
>>> from math import sin, cos, pi
>>> h = add_func(sin, cos)
>>> h(pi / 4)
1.414213562373095
```

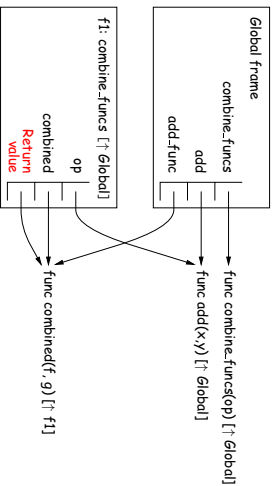
- What do the environments look like here? Think about it and try it out.

Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 14

The Environment Picture (I)

```
def combine_funcs(op):
    def combined(f, g):
        def val(x):
            return op(f(x),
                    g(x))
        return val
    return combined
add_func = combine_funcs(add)
combine_funcs(add)
```



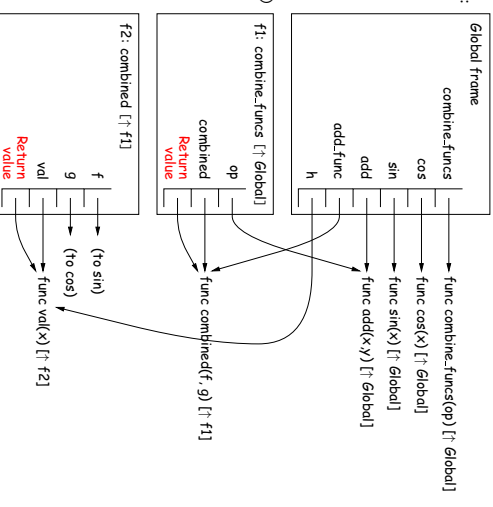
Legend: ↑ is short for "parent=".

Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 15

The Environment Picture (II)

```
def combine_funcs(op):
    def combined(f, g):
        def val(x):
            return op(f(x),
                    g(x))
        return val
    return combined
add_func = combine_funcs(add)
h = add_func(sin, cos)
```

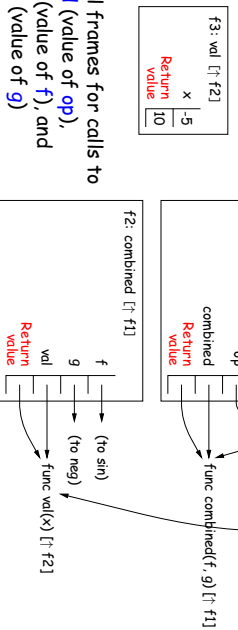


Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 16

The Environment Picture (III)

```
def combine_funcs(op):
    def combined(f, g):
        def val(x):
            return op(f(x), g(x))
        return val
    return combined
add_func = combine_funcs(add)
h = add_func(sin, cos)
h(-5)
```



- local frames for calls to
- `add` (value of `op`),
- `sin` (value of `f`), and
- `cos` (value of `g`)

Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 17

Generalize!

- Let's make a general function-combining function (that goes beyond addition):

```
>>> def combine_funcs(op):
...     """combine_funcs(OP)(f, g)(x) = OP(f(x), g(x))."""
...     def combined(f, g):
...         def val(x):
...             return op(f(x), g(x))
...         return val
...     return combined
```

- Now `add_func` itself can be constructed by a call to `combine_funcs`:

```
>>> from operator import add
>>> add_func = combine_funcs(add)
>>> from math import sin, cos, pi
>>> h = add_func(sin, cos)
>>> h(pi / 4)
1.414213562373095
```

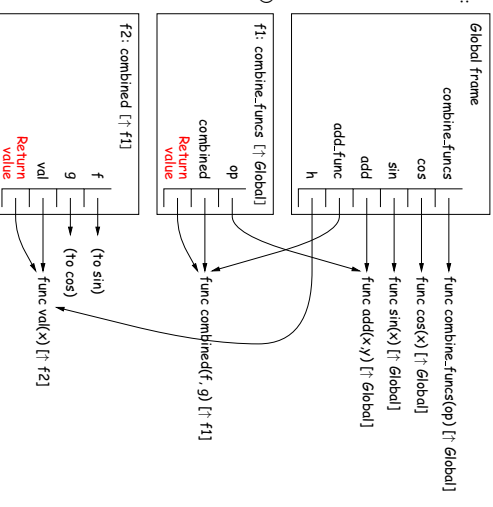
- What do the environments look like here? Think about it and try it out.

Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 14

The Environment Picture (II)

```
def combine_funcs(op):
    def combined(f, g):
        def val(x):
            return op(f(x),
                    g(x))
        return val
    return combined
add_func = combine_funcs(add)
h = add_func(sin, cos)
```



Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 16

A Fancy Example

- What does Python print, and why?

```
>>> def chain(n):
...     return lambda which: n if which else chain(n + 1)
>>> g1 = chain(1)
>>> g1(True)
1
>>> g2 = g1(False)
2
>>> g2(True)
3
>>> g2(False)
True
```

Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 18

A Fancy Example

- What does Python print, and why?

```
>>> def chain(n):
...     return lambda which: n if which else chain(n + 1)
>>> g1 = chain(1)
>>> g1(True)
1
>>> g2 = g1(False)
>>> g2
2
>>> g2(True)
1
>>> g2(False)(True)
```

Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 19

A Fancy Example

- What does Python print, and why?

```
>>> def chain(n):
...     return lambda which: n if which else chain(n + 1)
>>> g1 = chain(1)
>>> g1(True)
1
>>> g2 = g1(False)
>>> g2
2
>>> function chain...
>>> g2(True)
2
>>> g2(False)(True)
```

Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 20

A Fancy Example

- What does Python print, and why?

```
>>> def chain(n):
...     return lambda which: n if which else chain(n + 1)
>>> g1 = chain(1)
>>> g1(True)
1
>>> g2 = g1(False)
>>> g2
2
>>> function chain...
>>> g2(False)(True)
```

Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 21

A Fancy Example

- What does Python print, and why?

```
>>> def chain(n):
...     return lambda which: n if which else chain(n + 1)
>>> g1 = chain(1)
>>> g1(True)
1
>>> g2 = g1(False)
>>> g2
2
>>> function chain...
>>> g2(True)
2
>>> g2(False)(True)
```

Last modified: Sun Feb 19 01:42:48 2017

CS61A, Lecture #4 22