Lecture 4: Environment Diagrams

Brian Hou
June 21, 2016
Announcements
Announcements

• Homework 1 is due Sunday 6/26
Announcements

• Homework 1 is due Sunday 6/26
• Project 1 is released, due Thursday 6/30
Announcements

• Homework 1 is due Sunday 6/26
• Project 1 is released, due Thursday 6/30
  • Earn 1 EC point for completing it by Wednesday 6/29
Announcements

- Homework 1 is due Sunday 6/26
- Project 1 is released, due Thursday 6/30
  - Earn 1 EC point for completing it by Wednesday 6/29
- Go to discussion today! Each discussion is worth two exam recovery points
Announcements

• Homework 1 is due Sunday 6/26
• Project 1 is released, due Thursday 6/30
  • Earn 1 EC point for completing it by Wednesday 6/29
• Go to discussion today! Each discussion is worth two exam recovery points
• Ask questions during lecture on Piazza!
Roadmap

- Introduction
- Functions
- Data
- Mutability
- Objects
- Interpretation
- Paradigms
- Applications
This week (Introduction), the goals are:

- To learn the fundamentals of programming
- To become comfortable with Python
Abstraction
Abstraction

"The essence of abstraction is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context."

– John V. Guttag, *Introduction to Computation and Programming Using Python*
Discussion Question 1

What does pyramid compute?

```python
def pyramid(n):
    a, b, total = 0, n, 0
    while b:
        a, b = a+1, b-1
        total = total + a + b
    return total
```

$n^2$

$(n + 1)^2$

$2 \cdot (n + 1)$

$n^2 + 1$

$n \cdot (n + 1)$
Discussion Question 1

What does pyramid compute?

```python
def pyramid(n):
    a, b, total = 0, n, 0
    while b:
        a, b = a+1, b-1
        total = total + a + b
    return total
```

pyramid(4)
Discussion Question 1

What does `pyramid` compute?

```python
def pyramid(n):
    a, b, total = 0, n, 0
    while b:
        a, b = a+1, b-1
        total = total + a + b
    return total
```

`pyramid(4)`
Discussion Question 1

What does pyramid compute?

```python
def pyramid(n):
    a, b, total = 0, n, 0
    while b:
        a, b = a+1, b-1
        total = total + a + b
    return total
```

pyramid(4)

- $n^2$
- $(n+1)^2$
- $2 \cdot (n+1)$
- $n^2 + 1$
- $n \cdot (n+1)$
Discussion Question 1

What does pyramid compute?

```python
def pyramid(n):
    a, b, total = 0, n, 0
    while b:
        a, b = a + 1, b - 1
        total = total + a + b
    return total
```

\[ n^2 \]

\[ (n + 1)^2 \]

\[ 2 \cdot (n + 1) \]

\[ n^2 + 1 \]

\[ n \cdot (n + 1) \]

\[ \text{pyramid}(4) \]
Discussion Question 1

What does pyramid compute?

```python
def pyramid(n):
    a, b, total = 0, n, 0
    while b:
        a, b = a+1, b-1
        total = total + a + b
    return total
```

pyramid(4)
Discussion Question 1

What does pyramid compute?

```python
def pyramid(n):
    a, b, total = 0, n, 0
    while b:
        a, b = a+1, b-1
        total = total + a + b
    return total
```

pyramid(4)
What does pyramid compute?

```python
def pyramid(n):
    a, b, total = 0, n, 0
    while b:
        a, b = a+1, b-1
        total = total + a + b
    return total
```

pyramid(4)
def pyramid(n):
    a, b, total = 0, n, 0
    while b:
        a, b = a + 1, b - 1
        total = total + a + b
    return total

n²

(n + 1)²

2 · (n + 1)

n² + 1

n · (n + 1)

What does pyramid compute?

Discussion Question 1
What does pyramid compute?

```python
def pyramid(n):
    a, b, total = 0, n, 0
    while b:
        a, b = a+1, b-1
        total = total + a + b
    return total
```

\[
\begin{align*}
&n^2 \\
&(n+1)^2 \\
&2 \cdot (n+1) \\
&n^2 + 1 \\
&n \cdot (n+1)
\end{align*}
\]

pyramid(4)
Discussion Question 1

What does pyramid compute?

```python
def pyramid(n):
    a, b, total = 0, n, 0
    while b:
        a, b = a+1, b-1
        total = total + a + b
    return total
```

```latex
\begin{align*}
    & n^2 \\
    & (n+1)^2 \\
    & 2 \cdot (n+1) \\
    & n^2 + 1 \\
    & n \cdot (n+1)
\end{align*}
```

pyramid(4)
Discussion Question 1

What does pyramid compute?

```
def pyramid(n):
    a, b, total = 0, n, 0
    while b:
        a, b = a+1, b-1
        total = total + a + b
    return total
```

pyramid(4)

- $n^2$
- $(n + 1)^2$
- $2 \cdot (n + 1)$
- $n^2 + 1$
- $n \cdot (n + 1)$
Discussion Question 1

What does pyramid compute?

```
def pyramid(n):
    a, b, total = 0, n, 0
    while b:
        a, b = a+1, b-1
        total = total + a + b
    return total
```

pyramid(4)
Discussion Question 1

What does pyramid compute?

\[
\begin{align*}
\text{n}^2 & \\
(n + 1)^2 & \\
2 \cdot (n + 1) & \\
n^2 + 1 & \\
n \cdot (n + 1) & \\
\end{align*}
\]

```python
def pyramid(n):
    a, b, total = 0, n, 0
    while b:
        a, b = a + 1, b - 1
        total = total + a + b
    return total
```

\[
\begin{align*}
\text{square}(4) & \\
\end{align*}
\]
Tools for abstraction
Tools for abstraction

• Assignment is a simple form of abstraction: bind names to values
Tools for abstraction

• Assignment is a simple form of abstraction: bind names to values

• Function definition is a more powerful form of abstraction: bind names to a series of computations
Tools for abstraction

• Assignment is a simple form of abstraction: bind names to values

• Function definition is a more powerful form of abstraction: bind names to a series of computations

• *Functional abstraction* is the idea that we can call functions without thinking about how the function works
Tools for abstraction

- Assignment is a simple form of abstraction: bind names to values
- Function definition is a more powerful form of abstraction: bind names to a series of computations
- *Functional abstraction* is the idea that we can call functions without thinking about how the function works

\[
3 \xrightarrow{\text{square}} \ ?
\]
Tools for abstraction

• Assignment is a simple form of abstraction: bind names to values

• Function definition is a more powerful form of abstraction: bind names to a series of computations

• *Functional abstraction* is the idea that we can call functions without thinking about how the function works

```
3 ➩ square ➩ ?

4 ➩ square ➩ ?
```
Tools for abstraction

- Assignment is a simple form of abstraction: bind names to values
- Function definition is a more powerful form of abstraction: bind names to a series of computations
- Functional abstraction is the idea that we can call functions without thinking about how the function works

```
3 ▶ square ?
```

```
4 ▶ square ?
```
Tools for abstraction

• Assignment is a simple form of abstraction: bind names to values

• Function definition is a more powerful form of abstraction: bind names to a series of computations

• Functional abstraction is the idea that we can call functions without thinking about how the function works
Tools for abstraction

- Assignment is a simple form of abstraction: bind names to values
- Function definition is a more powerful form of abstraction: bind names to a series of computations
- Functional abstraction is the idea that we can call functions without thinking about how the function works
Tools for abstraction

- Assignment is a simple form of abstraction: bind names to values
- Function definition is a more powerful form of abstraction: bind names to a series of computations
- *Functional abstraction* is the idea that we can call functions without thinking about how the function works.
Tools for abstraction

- Assignment is a simple form of abstraction: bind names to values
- Function definition is a more powerful form of abstraction: bind names to a series of computations
- *Functional abstraction* is the idea that we can call functions without thinking about how the function works

```
3 → square → add → sqrt
4 → square → add → sqrt
```

5
Miscellaneous Python features

- Operators
- Multiple return values
- Docstrings
- Doctests
- Default arguments
Environment Diagrams
Lists and **for** Loops
Lists and **for** Loops

\[s = [3, 1, 4, 1, 5, 9]\]
Lists and **for** Loops

`s = [3, 1, 4, 1, 5, 9]
def max_difference(s):`
Lists and **for** Loops

```python
s = [3, 1, 4, 1, 5, 9]
def max_difference(s):
    smallest = s[0]
```

Lists and **for** Loops

```python
s = [3, 1, 4, 1, 5, 9]
def max_difference(s):
    smallest = s[0]
    largest = s[0]
```
Lists and **for** Loops

```python
s = [3, 1, 4, 1, 5, 9]

def max_difference(s):
    smallest = s[0]
    largest = s[0]
    for elem in s:
```

(Please note: the code snippet is incomplete as it is expected to continue with the implementation of the `max_difference` function.)
Lists and **for** Loops

```python
s = [3, 1, 4, 1, 5, 9]
def max_difference(s):
    smallest = s[0]
    largest = s[0]
    for elem in s:
        if elem < smallest:
            smallest = elem
```
Lists and **for** Loops

```python
s = [3, 1, 4, 1, 5, 9]

def max_difference(s):
    smallest = s[0]
    largest = s[0]
    for elem in s:
        if elem < smallest:
            smallest = elem
```
Lists and **for** Loops

```
s = [3, 1, 4, 1, 5, 9]
def max_difference(s):
    smallest = s[0]
    largest = s[0]
    for elem in s:
        if elem < smallest:
            smallest = elem
        if elem > largest:
```
Lists and **for** Loops

```python
s = [3, 1, 4, 1, 5, 9]
def max_difference(s):
    smallest = s[0]
    largest = s[0]
    for elem in s:
        if elem < smallest:
            smallest = elem
        if elem > largest:
            largest = elem
```
Lists and **for** Loops

```
s = [3, 1, 4, 1, 5, 9]
def max_difference(s):
    smallest = s[0]
    largest = s[0]
    for elem in s:
        if elem < smallest:
            smallest = elem
        if elem > largest:
            largest = elem
    return largest - smallest
```
Lists and **for** Loops

```python
s = [3, 1, 4, 1, 5, 9]
def max_difference(s):
    smallest = s[0]
    largest = s[0]
    for elem in s:
        if elem < smallest:
            smallest = elem
        if elem > largest:
            largest = elem
    return largest - smallest
max_difference(s)
```
Lists and **for** Loops

```python
s = [3, 1, 4, 1, 5, 9]
def max_difference(s):
    smallest = s[0]
    largest = s[0]
    for elem in s:
        if elem < smallest:
            smallest = elem
        if elem > largest:
            largest = elem
    return largest - smallest

max_difference(s)
```
Functions and **while** loops
Functions and **while** loops

x = 2
Functions and **while** loops

\[
x = 2
\]

```python
def repeated(f, n, x):
```

```python
def repeated(f, n, x):
```
Functions and **while** loops

```python
x = 2
def repeated(f, n, x):
    while n > 0:
```

Functions and **while** loops

```python
dx = 2
def repeated(f, n, x):
    while n > 0:
        x = f(x)
```

```python
x = 2
def repeated(f, n, x):
    while n > 0:
        x = f(x)
```
Functions and **while** loops

```
x = 2

def repeated(f, n, x):
    while n > 0:
        x = f(x)
        n -= 1
```
Functions and **while** loops

\[ x = 2 \]

```python
def repeated(f, n, x):
    while n > 0:
        x = f(x)
        n -= 1
    return x
```

Functions and **while** loops

```python
x = 2

def repeated(f, n, x):
    while n > 0:
        x = f(x)
        n -= 1
    return x

def square(x):
```

---

`x = 2`

```python
def repeated(f, n, x):
    while n > 0:
        x = f(x)
        n -= 1
    return x
def square(x):
```
Functions and **while** loops

x = 2

```python
def repeated(f, n, x):
    while n > 0:
        x = f(x)
        n -= 1
    return x

def square(x):
    return x * x
```
Functions and \textbf{while} loops

\begin{verbatim}
x = 2

def repeated(f, n, x):
    while n > 0:
        x = f(x)
        n -= 1
    return x

def square(x):
    return x * x

repeated(square, x, 3)
\end{verbatim}
x = 2

def repeated(f, n, x):
    while n > 0:
        x = f(x)
        n -= 1
    return x

def square(x):
    return x * x

repeated(square, x, 3)
Lambda Expressions
Lambda Expressions
Lambda Expressions

>>> x = 10
Lambda Expressions

```python
>>> x = 10

>>> square = x * x
```
Lambda Expressions

```python
>>> x = 10

>>> square = x * x

An expression: this one evaluates to a number
```
Lambda Expressions

```python
>>> x = 10

>>> square = x * x

An expression: this one evaluates to a number

>>> square = lambda x: x * x
```
Lambda Expressions

```python
>>> x = 10
An expression: this one evaluates to a number

>>> square = x * x
Also an expression: evaluates to a function

>>> square = lambda x: x * x
```

An expression: this one evaluates to a number

Also an expression: evaluates to a function
Lambda Expressions

```python
>>> x = 10  
An expression: this one evaluates to a number
```  
```python
>>> square = x * x  
Also an expression: evaluates to a function
```  
```python
>>> square = lambda x: x * x  
Important: No "return" keyword!
```
Lambda Expressions

```python
>>> x = 10
An expression: this one evaluates to a number

>>> square = x * x
Also an expression: evaluates to a function

>>> square = lambda x: x * x
A function
```
Lambda Expressions

```python
>>> x = 10

An expression: this one evaluates to a number

>>> square = x * x

Also an expression: evaluates to a function

>>> square = lambda x: x * x

Important: No "return" keyword!

A function
with parameter x
```
Lambda Expressions

```python
>>> x = 10
An expression: this one evaluates to a number

>>> square = x * x
Also an expression: evaluates to a function

>>> square = lambda x: x * x
Important: No "return" keyword!

A function
    with parameter x
    that returns the value of "x * x"
```
Lambda Expressions

```python
>>> x = 10

>>> square = x * x

An expression: this one evaluates to a number

>>> square = lambda x: x * x

Also an expression: evaluates to a function

Important: No "return" keyword!

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

>>> square(4)
```

An expression: this one evaluates to a number

Also an expression: evaluates to a function

Important: No "return" keyword!

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

>>> x = 10
An expression: this one evaluates to a number

>>> square = x * x
Also an expression: evaluates to a function

>>> square = lambda x: x * x
Important: No "return" keyword!

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

>>> square(4)
16
Lambda Expressions

```python
>>> x = 10
An expression: this one evaluates to a number

>>> square = x * x
Also an expression: evaluates to a function

>>> square = lambda x: x * x
Important: No "return" keyword!
A function with parameter x
that returns the value of "x * x"

>>> square(4)
16
Must be a single expression
```
Lambda Expressions

>>> x = 10

An expression: this one evaluates to a number

>>> square = x * x

Also an expression: evaluates to a function

>>> square = lambda x: x * x

Important: No "return" keyword!

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

>>> square(4)

16

Must be a single expression

Lambda expressions in Python cannot contain statements at all!
Lambda Expressions

```python
>>> x = 10
An expression: this one evaluates to a number

>>> square = x * x
Also an expression: evaluates to a function

>>> square = lambda x: x * x
Important: No "return" keyword!
A function with parameter x that returns the value of "x * x"

>>> square(4)
16
Must be a single expression
```

Lambda expressions in Python cannot contain statements at all!

Lambda expressions aren't common in Python, but important in general
lambda

x = 2

def repeated(f, n, x):
    while n > 0:
        x = f(x)
        n -= 1
    return x

def square(x):
    return x * x

repeated(square, x, 3)
```python
x = 2

def repeated(f, n, x):
    while n > 0:
        x = f(x)
        n -= 1
    return x

def square(x):
    return x * x

repeated(square, x, 3)
```

```python
x = 2

def repeated(f, n, x):
    while n > 0:
        x = f(x)
        n -= 1
    return x

square = lambda x: x * x

repeated(square, x, 3)
```
\[
\begin{align*}
\text{def} \ & \ \text{square}(x): \\
& \quad \text{return} \ x \ * \ x \\
\text{repeated}(\text{square}, \ x, \ 3)
\end{align*}
\]
lambda
lambda

Global frame
x 2
repeated
square

f1: repeated
  f
  n 0
  x 81
  Return value 81

f2: square
  x 3
  Return value 9

f3: square
  x 9
  Return value 81
lambda
lambda

Global frame
x | 2
repeated
square

func repeated(f, n, x)

func square(x)

f1: repeated
f
n 0
x 81
Return value 81

f2: square
x 3
Return value 9

f3: square
x 9
Return value 81

Global frame
x | 2
repeated
square

func repeated(f, n, x)

func \lambda(x)

f1: repeated
f
n 0
x 81
Return value 81

f2: \lambda
x 3
Return value 9

f3: \lambda
x 9
Return value 81
lambda