Higher-Order Functions
Announcements
Designing Functions
Describing Functions
Describing Functions

A function's *domain* is the set of all inputs it might possibly take as arguments.
Describing 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.
Describing 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.
Describing 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.

```python
def square(x):
    """Return X * X."""
```
Describing 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.

```python
def square(x):
    """Return X * X."""

    x is a number
```
Describing 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):
    """Return X * X."""

    x is a number

    square returns a non-negative real number
Describing 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):
    """Return X * X."""

    x is a number

    square returns a non-negative real number

    square returns the square of x
A Guide to Designing Function
A Guide to Designing Function

Give each function exactly one job, but make it apply to many related situations
A Guide to Designing Function

Give each function exactly one job, but make it apply to many related situations

```python
>>> round(1.23)
1
```
A Guide to Designing Function

Give each function exactly one job, but make it apply to many related situations

```python
>>> round(1.23)  # >>> round(1.23, 1)
1.2
```

A Guide to Designing Function

Give each function exactly one job, but make it apply to many related situations

```python
>>> round(1.23)      >>> round(1.23, 1)      >>> round(1.23, 0)
1                 1.2                 1
```
A Guide to Designing Function

Give each function exactly one job, but make it apply to many related situations

```python
>>> round(1.23)
1
>>> round(1.23, 1)
1.2
>>> round(1.23, 0)
1
>>> round(1.23, 5)
1.23
```
A Guide to Designing Function

Give each function exactly one job, but make it apply to many related situations

```python
>>> round(1.23)
1
>>> round(1.23, 1)
1.2
>>> round(1.23, 0)
1
>>> round(1.23, 5)
1.23
```

Don’t repeat yourself (DRY). Implement a process just once, but execute it many times.
A Guide to Designing Function

Give each function exactly one job, but make it apply to many related situations

```python
>>> round(1.23)  
1

>>> round(1.23, 1)  
1.2

>>> round(1.23, 0)  
1

>>> round(1.23, 5)  
1.23
```

Don’t repeat yourself (DRY). Implement a process just once, but execute it many times.
A Guide to Designing Function

Give each function exactly one job, but make it apply to many related situations

```python
>>> round(1.23)
1
>>> round(1.23, 1)
1.2
>>> round(1.23, 0)
1
>>> round(1.23, 5)
1.23
```

Don’t repeat yourself (DRY). Implement a process just once, but execute it many times.
Generalization
Generalizing Patterns with Arguments
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape: 

\[ r \]
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:

Area:
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:  

Area:  \( r^2 \)
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

**Shape:**
- Square: $r$
- Circle: $r$
- Hexagon: $r$

**Area:**
- Square: $r^2$
- Circle: $\pi \cdot r^2$
- Hexagon: $\pi \cdot r^2$
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:
- Square: $r$
- Circle: $r$
- Hexagon: $r$

Area:
- Square: $r^2$
- Circle: $\pi \cdot r^2$
- Hexagon: $\frac{3\sqrt{3}}{2} \cdot r^2$
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:

Area: \( 1 \cdot r^2 \) \( \pi \cdot r^2 \) \( \frac{3\sqrt{3}}{2} \cdot r^2 \)
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

**Shape:**
- Square: $r^2$
- Circle: $\pi \cdot r^2$
- Hexagon: $\frac{3\sqrt{3}}{2} \cdot r^2$
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:

Area:

\[1 \cdot r^2\]

\[\pi \cdot r^2\]

\[\frac{3\sqrt{3}}{2} \cdot r^2\]
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:

Area:
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:

Area:

Finding common structure allows for shared implementation
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:

Area:

Finding common structure allows for shared implementation

(Demo)
Higher-Order Functions
Generalizing Over Computational Processes
Generalizing Over Computational Processes

The common structure among functions may be a computational process, rather than a number.
Generalizing Over Computational Processes

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

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

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

\[
\sum_{k=1}^{5} \frac{8}{(4k - 3)(4k - 1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} = 3.04
\]
Generalizing Over Computational Processes

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

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

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

\[ \sum_{k=1}^{5} \frac{8}{(4k - 3) \cdot (4k - 1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} = 3.04 \]
Generalizing Over Computational Processes

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

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

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

\[
\sum_{k=1}^{5} \frac{8}{(4k - 3) \cdot (4k - 1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} = 3.04
\]
Generalizing Over Computational Processes

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

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

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

\[
\sum_{k=1}^{5} \frac{8}{(4k - 3) \cdot (4k - 1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} = 3.04
\]
Generalizing Over Computational Processes

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

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

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

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

(Demo)
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence.\n    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence."

    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total

# Local function definitions; returning functions

def make_adder(n):
    """Return a function that takes one argument k and returns k + n."
    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
    """
    def adder(k):
        return k + n
    return adder

def compose1(f, g):
    """Return a function that composes f and g."
    f, g −− functions of a single argument
    """
    def h(x):
        return f(g(x))
    return h
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence."
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total

>>> summation(5, cube)
225
"""
Summation Example

```python
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence."""
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```

```python
>>> summation(5, cube)
225
"""
```

Function of a single argument (not called "term")

A formal parameter that will be bound to a function

The function bound to term gets called here
Summation Example

def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence."

    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total

>>> summation(5, cube)
225
"""

Function of a single argument (not called "term")
A formal parameter that will be bound to a function
The cube function is passed as an argument value
The function bound to term gets called here
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence."

    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total

>>> summation(5, cube)
225

"""
Functions as Return Values
Locally Defined Functions
Locally Defined Functions

Functions defined within other function bodies are bound to names in a local frame
Locally Defined Functions

Functions defined within other function bodies are bound to names in a local frame

```python
def make_adder(n):
    '''Return a function that takes one argument k and returns k + n.'''

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

>>> add_three = make_adder(3)
>>> add_three(4)
7

```

Locally Defined Functions
Locally Defined Functions

Functions defined within other function bodies are bound to names in a local frame.

A function that returns a function

def make_adder(n):
    """Return a function that takes one argument k and returns k + n."
    def adder(k):
        return k + n
    return adder

>>> add_three = make_adder(3)
>>> add_three(4)
7
"""
Locally Defined Functions

Functions defined within other function bodies are bound to names in a local frame.

A function that returns a function

```python
def make_adder(n):
    """Return a function that takes one argument k and returns k + n."
    return lambda k: k + n

>>> add_three = make_adder(3)
>>> add_three(4)
7
"""
```

The name add_three is bound to a function.
Locally Defined Functions

Functions defined within other function bodies are bound to names in a local frame.

A function that returns a function

```python
def make_adder(n):
    """Return a function that takes one argument k and returns k + n."""
    def adder(k):
        return k + n
    return adder
```

The name add_three is bound to a function

```python
def add_three = make_adder(3)
>>> add_three(4)
7
```

A def statement within another def statement
Locally Defined Functions

Functions defined within other function bodies are bound to names in a local frame.

A function that returns a function

```python
def make_adder(n):
    """Return a function that takes one argument k and returns k + n."
    def adder(k):
        return k + n
    return adder
```

The name add_three is bound to a function

```
>>> add_three = make_adder(3)
>>> add_three(4)
7
```

A def statement within another def statement

Can refer to names in the enclosing function
Call Expressions as Operator Expressions
Call Expressions as Operator Expressions

make_adder(1) ( 2 )
Call Expressions as Operator Expressions

\[
\text{Operator} \\
\text{make_adder(1) ( 2 )}
\]
Call Expressions as Operator Expressions

```
 Operator    Operand

 make_adder(1) (       2       )
```
Call Expressions as Operator Expressions

An expression that evaluates to a function

Operator

Operand

make_adder(1) ( 2 )
Call Expressions as Operator Expressions

An expression that evaluates to a function

Operator

make_adder(1)     (         2         )

An expression that evaluates to its argument

Operand

2
Call Expressions as Operator Expressions

An expression that evaluates to a function

Operator

An expression that evaluates to its argument

Operand

make_adder(1) ( 2 )
Call Expressions as Operator Expressions

An expression that evaluates to a function

Operator

An expression that evaluates to its argument

Operand

make_adder(1)     (         2         )

make_adder(1)
Call Expressions as Operator Expressions

An expression that evaluates to a function

Operator

An expression that evaluates to its argument

Operand

make_adder(1)     (         2         )

make_adder(1)

func make_adder(n)
Call Expressions as Operator Expressions

An expression that evaluates to a function

An expression that evaluates to its argument

make_adder(1)   (   2   )

func make_adder(n)

1
Call Expressions as Operator Expressions

An expression that evaluates to a function

An expression that evaluates to its argument

\texttt{make_adder(1) \ ( \ 2 \ )}

\texttt{func make_adder(n)}

\texttt{1 \ make_adder(n):}

\texttt{make_adder(n):}
Call Expressions as Operator Expressions

An expression that evaluates to a function

An expression that evaluates to its argument

make_adder(1) ( 2 )

func make_adder(n)

1

make_adder( n ):
def adder(k):
    return k + n
return adder
Call Expressions as Operator Expressions

An expression that evaluates to a function

Operator

make_adder(1)

Operand

An expression that evaluates to its argument

func make_adder(n)

1

make_adder(n):

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

func adder(k)
Call Expressions as Operator Expressions

An expression that evaluates to a function

Operator

An expression that evaluates to its argument

Operand

func make_adder(n)

make_adder(1)

func adder(k)

1

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

func adder(k)
Call Expressions as Operator Expressions

- An expression that evaluates to a function
- An expression that evaluates to its argument

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

func adder(k)
```

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

func make_adder(1)
```
Call Expressions as Operator Expressions

An expression that evaluates to a function

Operator

make_adder(1) (2)

Operand

An expression that evaluates to its argument

func adder(k)

make_adder(1)

1

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

func adder(k)
Call Expressions as Operator Expressions

An expression that evaluates to a function

Operator

An expression that evaluates to its argument

Operand

func make_adder(n)

make_adder(1)

func adder(k)

1

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

func adder(k)

2

3

(2)
Lambda Expressions

(Demo)
Lambda Expressions
Lambda Expressions

```python
>>> x = 10
```
Lambda Expressions

```python
>>> x = 10

>>> square = x * x
```

Lambda Expressions

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

>>> square = x * x
```

Lambda Expressions

```python
>>> x = 10

>>> square = x * x

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

An expression: this one evaluates to a number
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
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
A function
    with formal 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
A function
with formal parameter x
that returns the value of "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
A function
with formal parameter x
that returns the value of "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 with formal parameter x that returns the value of "x * x"

Important: No "return" keyword!

Must be a single expression
```
Lambda Expressions

```python
>>> x = 10
>>> square = x * x
>>> square = lambda x: x * x
>>> square(4)
16
```

An expression: this one evaluates to a number

Also an expression: evaluates to a function

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

Important: No "return" keyword!

Must be a single expression
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
with formal parameter x
that returns the value of "x * x"

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

Lambda expressions are not common in Python, but important in general.

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
Important: No "return" keyword!
A function

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

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

Lambda expressions are not common in Python, but important in general.
Lambda expressions in Python cannot contain statements at all!
Lambda Expressions Versus Def Statements
Lambda Expressions Versus Def Statements

VS
Lambda Expressions Versus Def Statements

\[
square = \lambda x: x \times x
\]  

VS
Lambda Expressions Versus Def Statements

\[ \text{square} = \lambda x: x \times x \quad \text{VS} \quad \text{def square}(x): \text{return } x \times x \]
Lambda Expressions Versus Def Statements

\[
\text{square} = \lambda x: x \times x
\]

\[
\text{def square}(x):
\text{return } x \times x
\]

• Both create a function with the same domain, range, and behavior.
Lambda Expressions Versus Def Statements

\begin{align*}
\text{square} &= \lambda x : x \times x \\
\text{def square}(x) : \quad &\text{return } x \times x
\end{align*}

- Both create a function with the same domain, range, and behavior.
- Both bind that function to the name square.
Lambda Expressions Versus 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 bind that function to the name square.
- Only the def statement gives the function an intrinsic name, which shows up in environment diagrams but doesn't affect execution (unless the function is printed).
Lambda Expressions Versus Def Statements

\[ \text{square} = \text{lambda } x: x \times x \quad \text{VS} \quad \text{def square}(x): \text{return } x \times x \]

- Both create a function with the same domain, range, and behavior.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name, which shows up in environment diagrams but doesn't affect execution (unless the function is printed).

```
Global frame
square

func \lambda(x) <line 1> [parent=Global]
```

```
f1: \lambda <line 1> [parent=Global]
  x 4
  Return value 16
```
Lambda Expressions Versus Def Statements

\[
\text{square} = \lambda \ x: \ x \times x \quad \text{VS} \quad \text{def square}(x): \quad \text{return} \ x \times x
\]

- Both create a function with the same domain, range, and behavior.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name, which shows up in environment diagrams but doesn't affect execution (unless the function is printed).
Lambda Expressions Versus Def Statements

\[
square = \lambda x : x \times x
\]

vs

\[
def \text{square}(x):
    \text{return } x \times x
\]

- Both create a function with the same domain, range, and behavior.
- Both bind that function to the name \text{square}.
- Only the def statement gives the function an intrinsic name, which shows up in environment diagrams but doesn't affect execution (unless the function is printed).
Lambda Expressions Versus Def Statements

\[
square = \text{lambda } x: x \times x
\]

\[
def \text{square}(x):
  \text{return } x \times x
\]

- Both create a function with the same domain, range, and behavior.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name, which shows up in environment diagrams but doesn't affect execution (unless the function is printed).
Lambda Expressions Versus Def Statements

\[
square = \lambda x: x \times x
\]

\[
def square(x):
    return x \times x
\]

- Both create a function with the same domain, range, and behavior.
- Both bind that function to the name `square`.
- Only the `def` statement gives the function an intrinsic name, which shows up in environment diagrams but doesn't affect execution (unless the function is printed).