Announcements
Iteration Example
The Fibonacci Sequence
The Fibonacci Sequence
The Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987
The Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987
The Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987
The Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987
def fib(n):
    # Compute the nth Fibonacci number, for N >= 1.
    pred, curr = 0, 1  # 0th and 1st Fibonacci numbers
    k = 1  # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr

The Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1  # 0th and 1st Fibonacci numbers
    k = 1  # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr  # The next Fibonacci number is the sum of the current one and its predecessor
        k = k + 1
    return curr

The Fibonacci Sequence
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1  # 0th and 1st Fibonacci numbers
    k = 1              # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, (pred + curr)
        k = k + 1
    return curr

The next Fibonacci number is the sum of the current one and its predecessor.
The Fibonacci Sequence

The next Fibonacci number is the sum of the current one and its predecessor.
The Fibonacci Sequence

<table>
<thead>
<tr>
<th>fib</th>
<th>pred</th>
<th>curr</th>
<th>n</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

```python
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1  # 0th and 1st Fibonacci numbers
    k = 1  # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, (pred + curr)
        k = k + 1
    return curr
```

The next Fibonacci number is the sum of the current one and its predecessor.
The Fibonacci Sequence

```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1  # 0th and 1st Fibonacci numbers
    k = 1              # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

The next Fibonacci number is the sum of the current one and its predecessor

The Fibonacci Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987
The Fibonacci Sequence

def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1  # 0th and 1st Fibonacci numbers
    k = 1  # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr

The next Fibonacci number is the sum of the current one and its predecessor

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987
The Fibonacci Sequence

```python
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1  # 0th and 1st Fibonacci numbers
    k = 1  # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

The next Fibonacci number is the sum of the current one and its predecessor.
The Fibonacci Sequence

```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1  # 0th and 1st Fibonacci numbers
    k = 1              # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr  # The next Fibonacci number is the sum of the current one and its predecessor
        k = k + 1
    return curr
```

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987
def fib(n):
    """Compute the nth Fibonacci number?"""
    pred, curr = 0, 1
    k = 1
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```python
def fib(n):
    """Compute the nth Fibonacci number?""
    pred, curr = 0, 1
    k = 1
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987
Discussion Question

Is this alternative definition of \texttt{fib} the same or different from the original \texttt{fib}? Does this alternative definition of \texttt{fib} produce the same sequence of values, or is it different from \texttt{fib}? The sequence produced by the original definition is:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987...

The sequence produced by the alternative definition is:

1, 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987...

```python
def \texttt{fib}(n):
    pred, curr = 0, 1
    while n > 0:
        k = 1
        pred, curr = curr, pred + curr
        k += 1
    return curr
```
Discussion Question

Is this alternative definition of \texttt{fib} the same or different from the original \texttt{fib}?

\begin{verbatim}
def fib(n):
    """Compute the nth Fibonacci number?""
    pred, curr = 0, 1
    k = 1
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
\end{verbatim}

\texttt{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377}
Discussion Question

Is this alternative definition of `fib` the same or different from the original `fib`?

```python
def fib(n):
    """Compute the nth Fibonacci number?""
    pred, curr = 0, 1
    k = 1
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377
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

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

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**

| def square(x):
| """Return x * x."""
| A function's *domain* is the set of all inputs it might possibly take as arguments.

| def fib(n):
| """Compute the nth Fibonacci number, for N >= 1."""
| 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.
**Describing Functions**

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

\[ x \text{ is a real number} \]

A function's **range** is the set of output values it might possibly return.

\[ \text{returns a non-negative real number} \]

A pure function's **behavior** is the relationship it creates between input and output.

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

def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
```
## Describing Functions

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

\[ x \text{ is a real number} \]

A function's **range** is the set of output values it might possibly return.

\[ \text{returns a non-negative real number} \]

A pure function's **behavior** is the relationship it creates between input and output.

\[ \text{return value is the square of the input} \]

---

**def square(x):**

"Return X * X."

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

**def fib(n):**

"""Compute the nth Fibonacci number, for N >= 1."""

```python
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
```
### Describing Functions

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

- \( x \) is a real number

A function's **range** is the set of output values it might possibly return.

- \( \text{returns a non-negative real number} \)

A pure function's **behavior** is the relationship it creates between input and output.

- \( \text{return value is the square of the input} \)

```python
def square(x):
    '''Return X * X.'''
    x is a real number
    returns a non-negative real number
    return value is the square of the input
```

```python
def fib(n):
    '''Compute the nth Fibonacci number, for N >= 1.'''
    n is an integer greater than or equal to 1
```

```
## Describing Functions

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

- **def square(x):**
  
  """Return X * X."""

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

  - *x* is a real number

A function's **range** is the set of output values it might possibly return.

- **def fib(n):**
  
  """Compute the nth Fibonacci number, for N >= 1."""

  A function's *range* is the set of output values it might possibly return.

  - *n* is an integer greater than or equal to 1

A pure function's **behavior** is the relationship it creates between input and output.

- **def square(x):**
  
  """Return X * X."""

  A pure function's *behavior* is the relationship it creates between input and output.

  - return value is the square of the input

- **def fib(n):**
  
  """Compute the nth Fibonacci number, for N >= 1."""

  A pure function's *behavior* is the relationship it creates between input and output.

  - returns a non-negative real number

  returns a Fibonacci number
### Describing Functions

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

- **def square(x):**
  
  """Return X * X."""

-  **x is a real number**

A function's **range** is the set of output values it might possibly return.

-  **def fib(n):**
  
  """Compute the nth Fibonacci number, for N >= 1."""

-  **n is an integer greater than or equal to 1**

A pure function's **behavior** is the relationship it creates between input and output.

-  **def square(x):**
  
  """Return X * X."""

-  **x is a real number**

-  **return value is the square of the input**

-  **def fib(n):**
  
  """Compute the nth Fibonacci number, for N >= 1."""

-  **returns a Fibonacci number**

-  **n is an integer greater than or equal to 1**

-  **return value is the nth Fibonacci number**
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               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)
1
>>> round(1.23, 1)
1.2
>>> round(1.23, 0)
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

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

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

(Demo)
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.

(Demo)
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:
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:

Area:

\[ r^2 \]

\[ \pi \cdot r^2 \]
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:

- Square: $1 \cdot r^2$
- 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:

\[ \begin{align*}
\text{Square:} & \quad r^2 \\
\text{Circle:} & \quad \pi \cdot r^2 \\
\text{Hexagon:} & \quad \frac{3\sqrt{3}}{2} \cdot r^2
\end{align*} \]
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:

Area:

\[
\begin{align*}
\text{Shape:} & \quad \text{Area:} \\
\text{Square} & \quad \frac{1}{4} \cdot r^2 \\
\text{Circle} & \quad \pi \cdot r^2 \\
\text{Hexagon} & \quad \frac{3\sqrt{3}}{2} \cdot r^2
\end{align*}
\]
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:

- \[ \text{Area: } 1 \cdot r^2 \]
- \[ \text{Area: } \pi \cdot r^2 \]
- \[ \text{Area: } \frac{3\sqrt{3}}{2} \cdot r^2 \]
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.

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.

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

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

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

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

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

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

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

```python
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
```
Summation Example

```python
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

>>> summation(5, cube)
225

# 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
```

A function of a single argument (not called "term")

A formal parameter that will be bound to a function
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

>>> 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

```python
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
```

A formal parameter that will be bound to a function

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

The cube function is passed as an argument value

The function bound to term gets called here

A formal parameter that will be bound to a function
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

>>> 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
Functions as Return Values

(Demo)
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."

    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
    
    def adder(k):
        return k + n
    return adder
```
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."

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

def adder(k):
    return k + n

return adder
```
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
>>> 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.
**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
```

The name `add_three` is bound to a function.

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

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

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

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

An expression that evaluates to a function

Operator

make_adder(1)     (         2         )

Operand
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 )
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(n)

func make_adder(n)
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)

1

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

func make_adder(n)

make_adder(1)

make_adder(1):

1

make_adder(n)
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)

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

Operand

An expression that evaluates to its argument

make_adder(1)     (         2         )

func make_adder(n)

make_adder(1)

1

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

An expression that evaluates to a function

An expression that evaluates to its argument

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

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

func make_adder(n)

make_adder(1)

func adder(k)

make_adder(1)

func adder(k):
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

`make_adder(1)`     (         2         )

Operator
Operand

`func adder(k)`

`make_adder(1)`

`func make_adder(n)`

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

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

func adder(k)

make_adder(n):

make_adder(1) ( 2 )

func make_adder(1)

func adder(k)

1

3