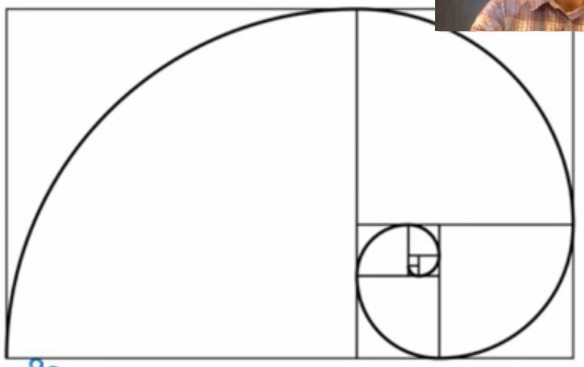
Higher-Order Functions

Announcements

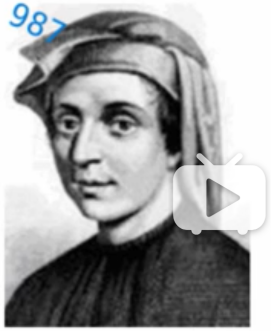
Iteration Example

The Fibonacci Sequence

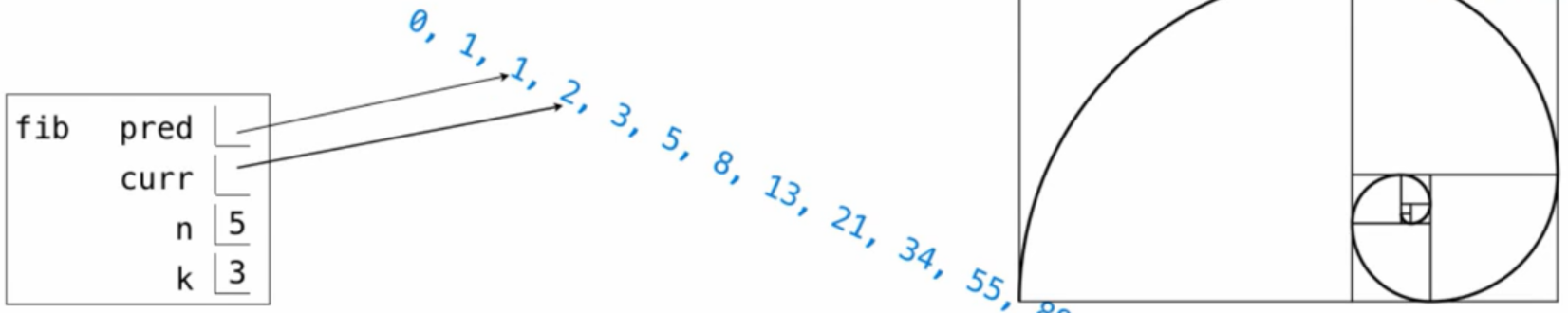
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987



```
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
```

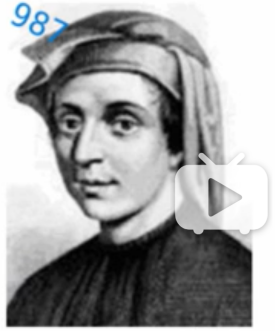


The Fibonacci Sequence




```
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



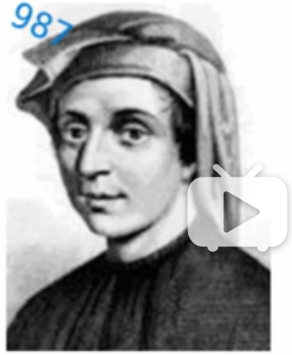
Discussion Question

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

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

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

Ans: a better version with the 0th element: n=0, curr=0, return curr



Designing Functions

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

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

```
>>> 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.

Define functions generally.

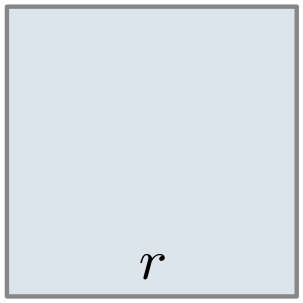
(Demo)

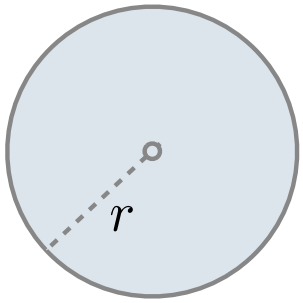
Generalization

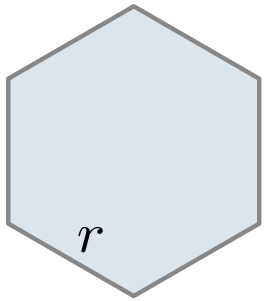
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:







Area: $1 \cdot r^2$ $\pi \cdot r^2$ $\frac{3\sqrt{3}}{2} \cdot r^2$

Finding common structure allows for shared implementation

(Demo)

Generalizing Patterns with Arguments

```
"""Generalization."""

from math import pi, sqrt

def area_square(r):
    assert r > 0, 'A length must be positive'
    return r * r

def area_circle(r):
    return r * r * pi

def area_hexagon(r):
    return r * r * 3 * sqrt(3) / 2
```

What is assert?

```
>>> assert 3 > 2, 'Math is broken'
>>> assert 2 > 3, 'That is false'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: That is false
```

Generalizing Patterns with Arguments

```
"""Generalization."""

from math import pi, sqrt

def area_square(r):
    assert r > 0, 'A length must be positive'
    return r * r

def area_circle(r):
    return r * r * pi

def area_hexagon(r):
    return r * r * 3 * sqrt(3) / 2
```

```
"""Generalization."""

from math import pi, sqrt

def area(r, shape_constant):
    assert r > 0, 'A length must be positive'
    return r * r * shape_constant

def area_square(r):
    return area(r, 1)

def area_circle(r):
    return area(r, pi)

def area_hexagon(r):
    return area(r, 3 * sqrt(3) / 2)
```

Higher-Order Functions

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)

Summation Example

```
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
```

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

0 + 1 + 8 + 27 + 64 + 125

The function bound to term gets called here

Higher-order function: a function that takes another function as an argument

Functions as Return Values

(Demo)

Locally Defined Functions

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

A function that returns a function

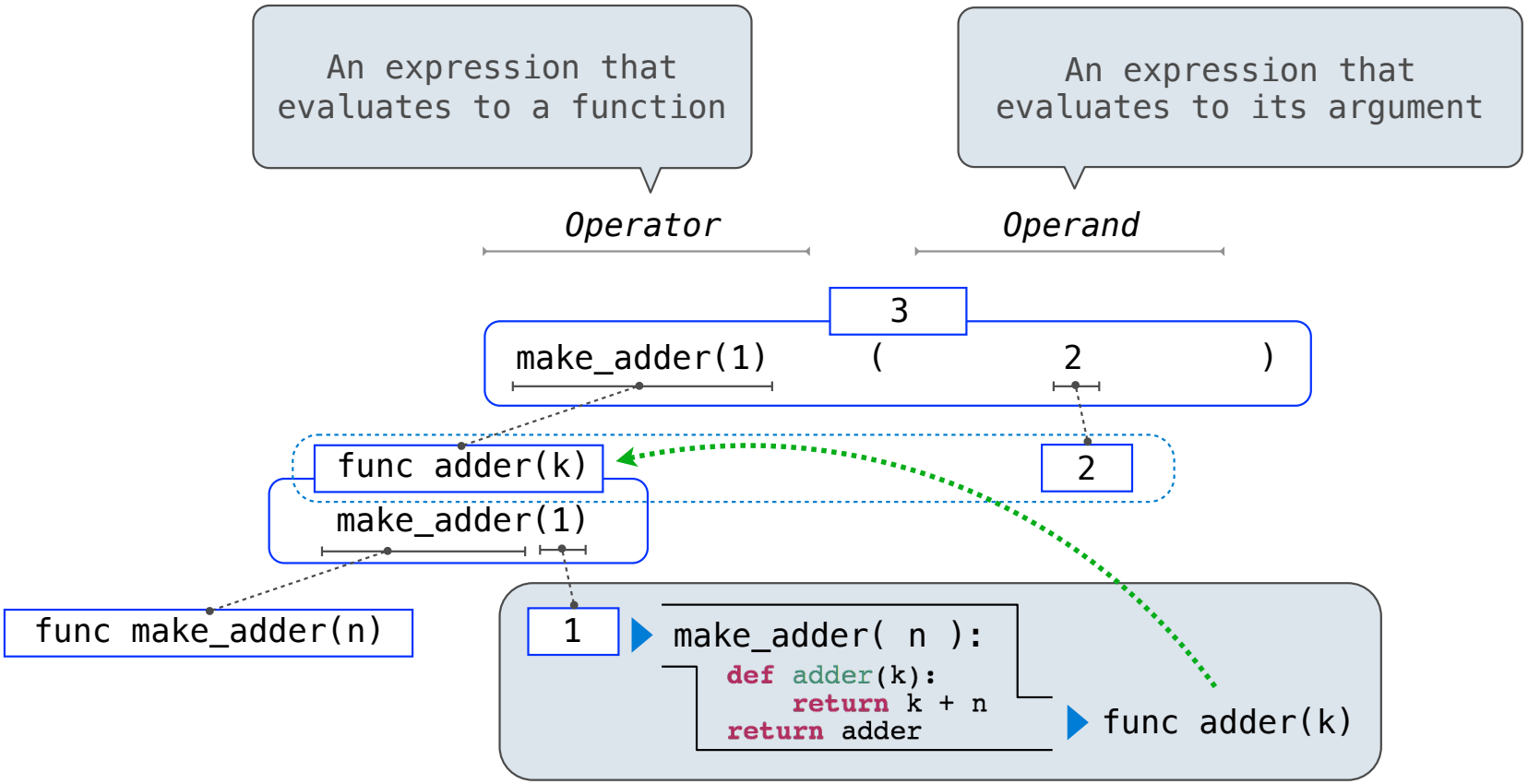
```
def make_adder(n):
    """Return a function that takes one argument k and returns k + n.

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



Lambda Expressions

(Demo)

Lambda Expressions

```
>>> 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

Important: No "return" keyword!

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

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 x: x * x) (4)
16
```

Lambda Expressions Versus Def Statements

lambda doesn't have a name until the assignment statement give it one

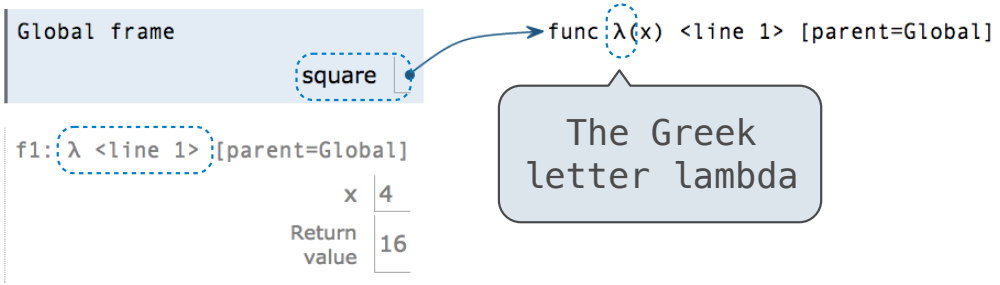
```
square = lambda x: x * x
```

VS

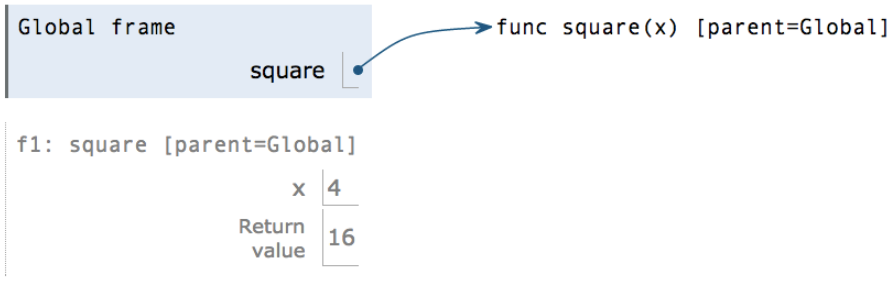
def has a name when it's created

```
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).




```
>>> square = lambda x: x * x
>>> square
<function <lambda> at 0x1003c1bf8>
```




```
>>> def square(x):
...     return x * x
...
>>> square
<function square at 0x10293e730>
```

intrinsic name