String Representations

String Representations

An object value should behave like the kind of data it is meant to represent

For instance, by producing a string representation of itself

Strings are important: they represent language and programs

In Python, all objects produce two string representations:

- The **str** is legible to humans
- The **repr** is legible to the Python interpreter

The **str** and **repr** strings are often the same, but not always

The repr String for an Object

The **repr** function returns a Python expression (a string) that evaluates to an equal object

```
repr(object) -> string

Return the canonical string representation of the object.
For most object types, eval(repr(object)) == object.
```

The result of calling **repr** on a value is what Python prints in an interactive session

```
>>> 12e12
12000000000000.0
>>> print(repr(12e12))
12000000000000.0
```

Some objects do not have a simple Python-readable string

```
>>> repr(min)
'<built-in function min>'
```

The str String for an Object

Human interpretable strings are useful as well:

```
>>> from fractions import Fraction
>>> half = Fraction(1, 2)
>>> repr(half)
'Fraction(1, 2)'
>>> str(half)
'1/2'
```

The result of calling **str** on the value of an expression is what Python prints using the **print** function:

```
>>> print(half)
1/2
```

(Demo)

Polymorphic Functions

Polymorphic Functions

Polymorphic function: A function that applies to many (poly) different forms (morph) of data

str and **repr** are both polymorphic; they apply to any object

repr invokes a zero-argument method __repr__ on its argument

```
>>> half.__repr__()
'Fraction(1, 2)'
```

str invokes a zero-argument method __str__ on its argument

```
>>> half.__str__()
'1/2'
```

Implementing repr and str

The behavior of **repr** is slightly more complicated than invoking `__repr__` on its argument:

- An instance attribute called `__repr__` is ignored! Only class attributes are found
- *Question*: How would we implement this behavior?

```
def repr(x):
    return x.__repr__(x)
```

```
def repr(x):
    return x.__repr__()
```

```
def repr(x):
    return type(x).__repr__(x)
```

```
def repr(x):
    return type(x).__repr__()
```

```
def repr(x):
    return super(x).__repr__()
```

The behavior of **str** is also complicated:

- An instance attribute called `__str__` is ignored
- If no `__str__` attribute is found, uses **repr** string
- (By the way, **str** is a class, not a function)
- *Question*: How would we implement this behavior?

(Demo)

Interfaces

Message passing: Objects interact by looking up attributes on each other (passing messages)

The attribute look-up rules allow different data types to respond to the same message

A **shared message** (attribute name) that elicits similar behavior from different object classes is a powerful method of abstraction

An interface is a set of shared messages, along with a specification of what they mean

Example:

Classes that implement `__repr__` and `__str__` methods that return Python-interpretable and human-readable strings implement an interface for producing string representations

(Demo)

Special Method Names

Special Method Names in Python

Certain names are special because they have built-in behavior

These names always start and end with two underscores

`__init__`	Method invoked automatically when an object is constructed
`__repr__`	Method invoked to display an object as a Python expression
`__add__`	Method invoked to add one object to another
`__bool__`	Method invoked to convert an object to True or False
`__float__`	Method invoked to convert an object to a float (real number)

```
>>> zero, one, two = 0, 1, 2
>>> one + two
3
>>> bool(zero), bool(one)
(False, True)
```

Same behavior using methods

```
>>> zero, one, two = 0, 1, 2
>>> one.__add__(two)
3
>>> zero.__bool__(), one.__bool__()
(False, True)
```

Special Methods

Adding instances of user-defined classes invokes either the `__add__` or `__radd__` method

```
>>> Ratio(1, 3) + Ratio(1, 6)
Ratio(1, 2)

>>> Ratio(1, 3).__add__(Ratio(1, 6))
Ratio(1, 2)

>>> Ratio(1, 6).__radd__(Ratio(1, 3))
Ratio(1, 2)
```

http://getpython3.com/diveintopython3/special-method-names.html

http://docs.python.org/py3k/reference/datamodel.html#special-method-names

(Demo)

Generic Functions

A polymorphic function might take two or more arguments of different types

Type Dispatching: Inspect the type of an argument in order to select behavior

Type Coercion: Convert one value to match the type of another

```
>>> Ratio(1, 3) + 1
Ratio(4, 3)

>>> 1 + Ratio(1, 3)
Ratio(4, 3)

>>> from math import pi
>>> Ratio(1, 3) + pi
3.4749259869231266
```

(Demo)

Generic Functions

Adding instances of user-defined classes invokes either the `__add__` or `__radd__` method

A polymorphic function might take two or more arguments of different types
Type Dispatching: Inspect the type of an argument in order to select behavior
Type Coercion: Convert one value to match the type of another

```
>>> Ratio(1, 3) + Ratio(1, 6)
Ratio(1, 2)

>>> Ratio(1, 3).__add__(Ratio(1, 6))
Ratio(1, 2)

>>> Ratio(1, 6).__radd__(Ratio(1, 3))
Ratio(1, 2)

>>> Ratio(1, 3) + pi
3.4749259869231266
```

```
>>> Ratio(1, 3) + 1
Ratio(4, 3)

>>> 1 + Ratio(1, 3)
Ratio(4, 3)
```

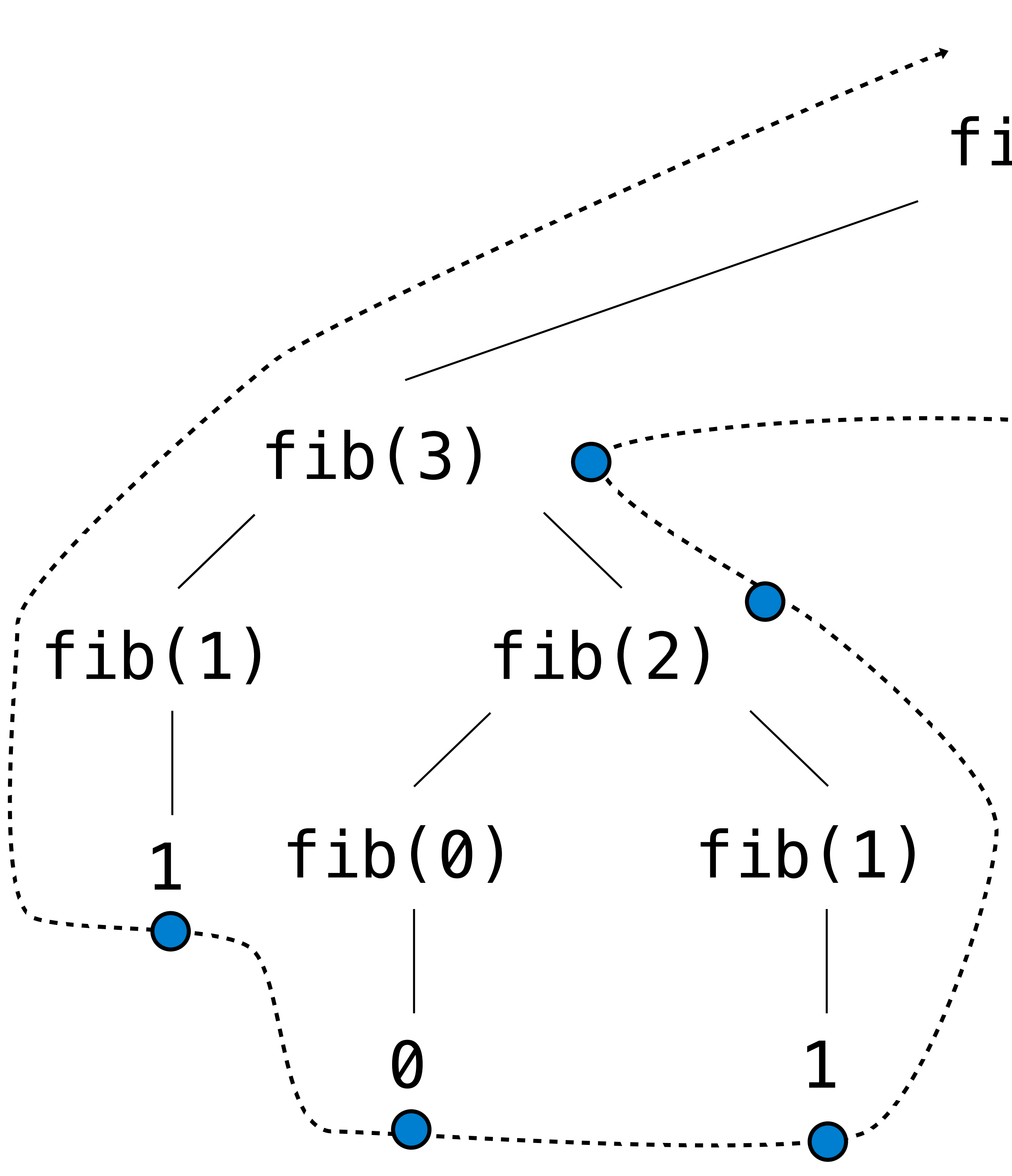
http://getpython3.com/diveintopython3/special-method-names.html
http://docs.python.org/py3k/reference/datamodel.html#special-method-names

(Demo)

Measuring Efficiency

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:



(Demo)

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

http://en.wikipedia.org/wiki/File:Fibonacci.jpg

Memoization

Memoization

Idea: Remember the results that have been computed before

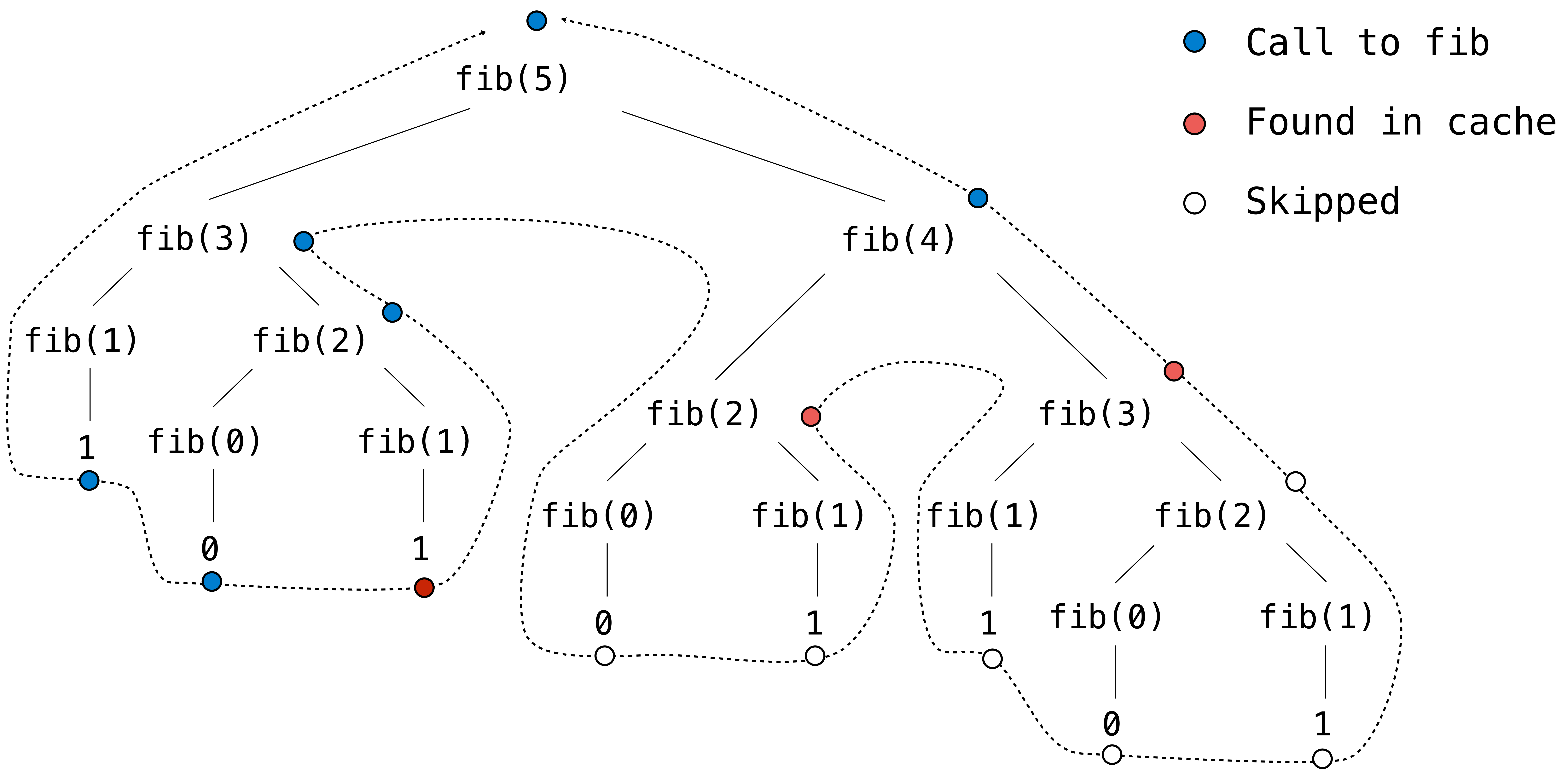
```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Keys are arguments that map to return values

Same behavior as f, if f is a pure function

(Demo)

Memoized Tree Recursion



Space

The Consumption of Space

Which environment frames do we need to keep during evaluation?

At any moment there is a set of active environments

Values and frames in active environments consume memory

Memory that is used for other values and frames can be recycled

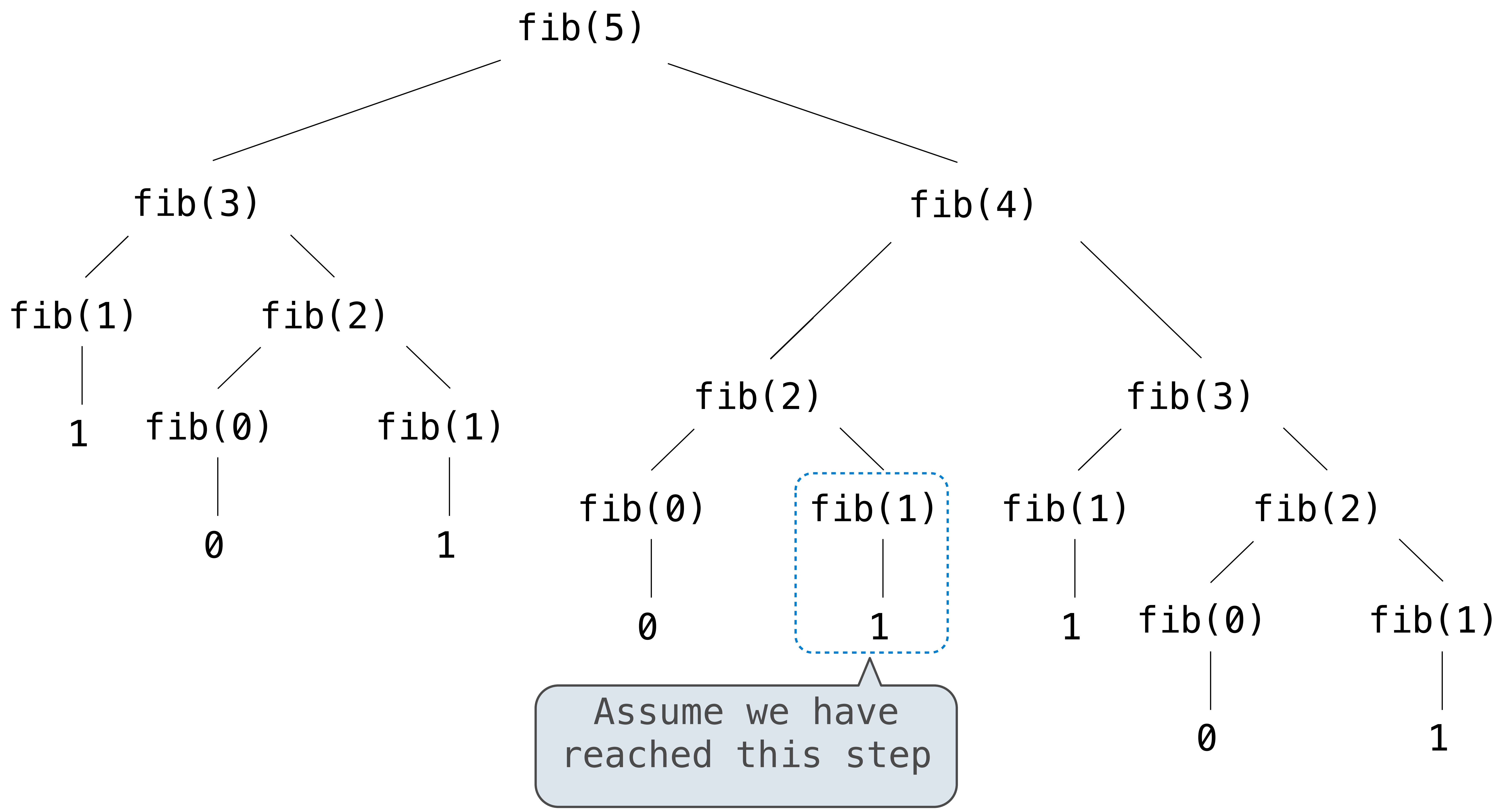
Active environments:

- Environments for any function calls currently being evaluated
- Parent environments of functions named in active environments

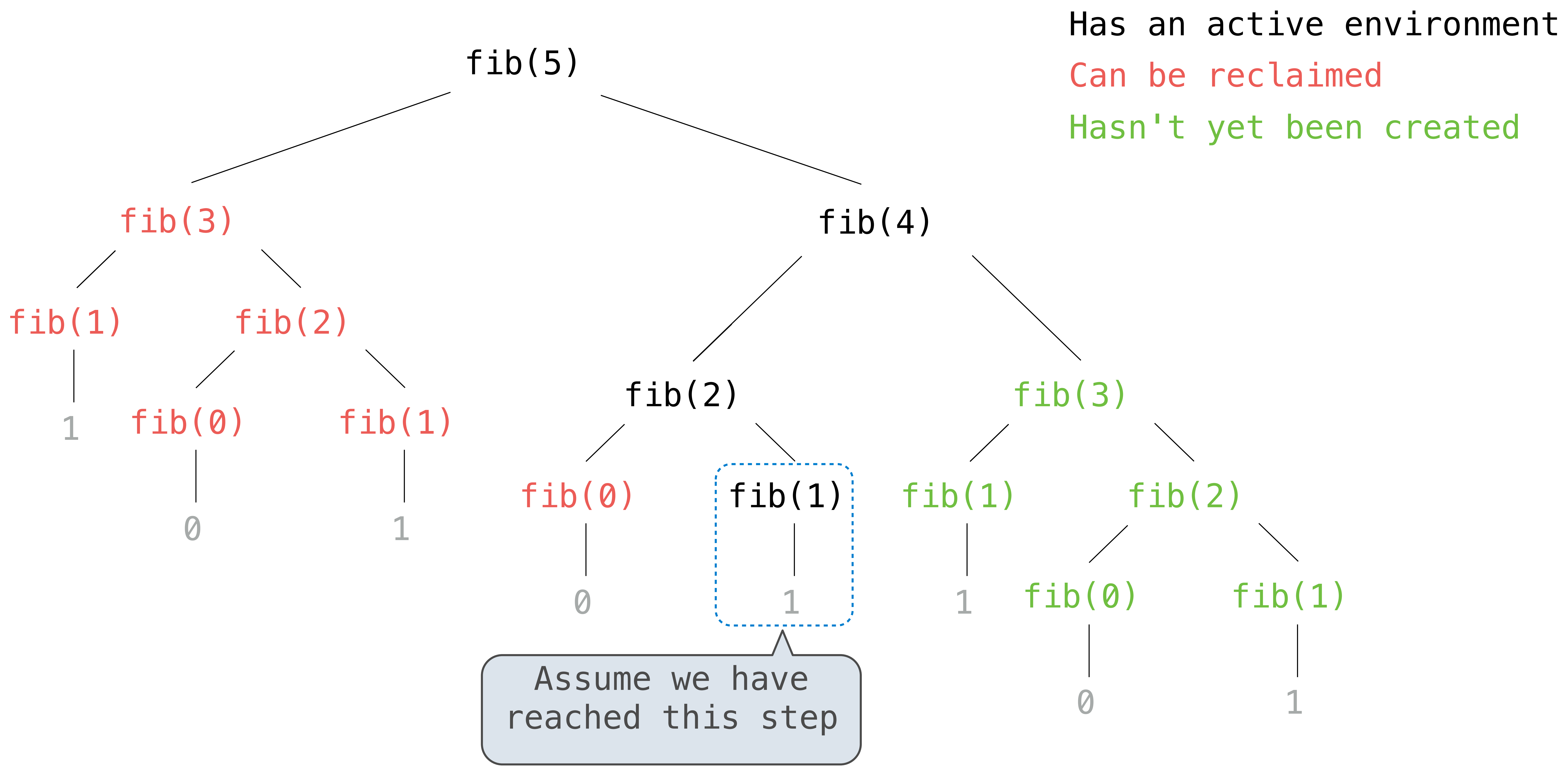
(Demo)

pythontutor.com/composingprograms.html#code=def%20fib%28n%29%3A%0A%20%20%20%20if%20n%20%3D%3D%200%20or%20n%20%3D%3D%201%3A%0A%20%20%20%20%20%20%20%20return%20n%0A%20%20%20%20else%3A%0A%20%20%20%20%20%20%20%20return%20fib%28n-2%29%20%2B%20fib%28n-1%29%0A%20%20%20%20%20%20%20%20%0Afib%286%2

Fibonacci Space Consumption



Fibonacci Space Consumption



Time

Comparing Implementations

Implementations of the same functional abstraction can require different resources

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

	Time (number of divisions)
Slow: Test each k from 1 through n	n
Fast: Test each k from 1 to square root n For every k, n/k is also a factor!	Greatest integer less than $\sqrt{n}$

Question: How many time does each implementation use division? (Demo)

Orders of Growth

Order of Growth

A method for bounding the resources used by a function by the "size" of a problem

n: size of the problem

R(n): measurement of some resource used (time or space)

$$R(n) = \Theta(f(n))$$

means that there are positive constants $\mathbf{k_1}$ and $\mathbf{k_2}$ such that

$$k_1 \cdot f(n) \leq \boxed{R(n)} \leq k_2 \cdot f(n)$$

for all **n** larger than some minimum **m**

Order of Growth of Counting Factors

Implementations of the same functional abstraction can require different amounts of time

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

	Time	Space
Slow: Test each k from 1 through n	$\Theta(n)$	$\Theta(1)$
Fast: Test each k from 1 to square root n For every k, n/k is also a factor!	$\Theta(\sqrt{n})$	$\Theta(1)$

Assumption: integers occupy a fixed amount of space

(Demo)

Exponentiation

Exponentiation

Goal: one more multiplication lets us double the problem size

```
def exp(b, n):
    if n == 0:
        return 1
    else:
        return b * exp(b, n-1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{otherwise} \end{cases}$$

```
def square(x):
    return x*x

def exp_fast(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(exp_fast(b, n//2))
    else:
        return b * exp_fast(b, n-1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ (b^{\frac{1}{2}n})^2 & \text{if } n \text{ is even} \\ b \cdot b^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

(Demo)

Exponentiation

Goal: one more multiplication lets us double the problem size

```
def exp(b, n):
    if n == 0:
        return 1
    else:
        return b * exp(b, n-1)


def square(x):
    return x*x

def exp_fast(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(exp_fast(b, n//2))
    else:
        return b * exp_fast(b, n-1)
```

Time	Space
$\Theta(n)$	$\Theta(n)$
$\Theta(\log n)$	$\Theta(\log n)$

Comparing Orders of Growth

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

```
def overlap(a, b):
    count = 0
    for item in a:            # Outer: length of a
        if item in b:         # Inner: length of b
            count += 1
    return count
```

If a and b are both length **n**, then overlap takes $\Theta(n^2)$ steps

Lower-order terms: The fastest-growing part of the computation dominates the total

$$\Theta(n^2) \qquad \Theta(n^2 + n) \qquad \Theta(n^2 + 500 \cdot n + \log_2 n + 1000)$$

Comparing orders of growth (n is the problem size)

$\Theta(b^n)$ Exponential growth. Recursive `fib` takes

$\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor

$\Theta(n^2)$ Quadratic growth. E.g., `overlap`

Incrementing n increases R(n) by the problem size n

$\Theta(n)$ Linear growth. E.g., slow `factors` or `exp`

$\Theta(\sqrt{n})$ Square root growth. E.g., `factors_fast`

$\Theta(\log n)$ Logarithmic growth. E.g., `exp_fast`

Doubling the problem only increments R(n).

$\Theta(1)$ Constant. The problem size doesn't matter