COMP7023 Lecture Notes (Weeks 05–08): Building Abstractions with Data

Adapted from *Composing Programs*[*] by John DeNero

Contents

[*] https://www.composingprograms.com/

1 Introduction

In Weeks 01–04, we studied computational processes and the role of functions in program design:

- Combining primitive data & operations.
- Higher-order functions.

In Weeks 05–08, we turn our attention to **data**: how to represent and manipulate information from many domains. Effective use of built-in and user-defined data types is essential for real-world computation, especially in data processing applications.

1.1 Native Data Types

Every Python value has a *class* defining its type and behavior. Values of the same class behave similarly. Integers, for example, are instances of `int`, and can be negated, added, etc. The built-in `type` function reveals a value's class:

```
>>> type(2)
<class 'int'>
```

Native (built-in) types have:

- Expressions (literals) that evaluate to their values.
- Built-in functions and operators for manipulation.

Integers.

`int` represents integers exactly, with no size limit.

```
>>> 12 + 3000000000000000000000000
3000000000000000000000012
```

Numeric types.

Python provides three native numeric types: `int`, `float`, and `complex`.

```
>>> type(1.5)
<class 'float'>
>>> type(1+1j)
<class 'complex'>
```

Floats.

Real numbers are stored in "floating point" form. `float` values approximate real numbers and have limited precision (i.e. they are "numbers with decimals"):

```
>>> 7 / 3 * 3
7.0
>>> 1 / 3 * 7 * 3
6.999999999999999
```

Division of two `int` values produces a `float`:

```
>>> type(1/3)
<class 'float'>
>>> 1/3
0.3333333333333333
```

Equality tests can be misleading due to approximation:

```
>>> 1/3 == 0.333333333333333312345  # Beware of float approximation
True
```

These differences between `int` and `float` have important implications in programming, though they are standardized (IEEE 754).

Non-numeric types.

Python also includes types such as `bool` for logical values:

```
True, False
```

Other native types (e.g. strings, lists) and user-defined types extend these principles, forming the foundation for data abstraction.

2 Data Abstraction

Key ideas:

- Most real objects have **compound structure** (e.g., position = latitude + longitude).
- **Data abstraction** separates:
  - *Representation* – how data are built.
  - *Usage* – how data are used.
- Improves modularity: change representation without changing behavior.
- Analogy: functional abstraction (hide implementation, keep interface).

2.1 Example: Rational Numbers

Goal: exact representation of fractions.

```
>>> 1/3
0.3333333333333333
>>> 1/3 == 0.333333333333333300000
True
```

Abstract interface:

```
rational(n, d)  # constructor
numer(x)        # numerator
denom(x)        # denominator
```

Wishful thinking: define operations assuming these exist.

```
>>> def add_rationals(x, y):
...     nx, dx = numer(x), denom(x)
...     ny, dy = numer(y), denom(y)
...     return rational(nx * dy + ny * dx, dx * dy)

>>> def mul_rationals(x, y):
...     return rational(numer(x)*numer(y), denom(x)*denom(y))

>>> def print_rational(x):
...     print(numer(x), '/', denom(x))

>>> def rationals_are_equal(x, y):
...     return numer(x)*denom(y) == numer(y)*denom(x)
```

2.2 Pairs

Concrete level: represent pairs using lists.

```
>>> pair = [10, 20]
>>> x, y = pair
>>> x, y
(10, 20)
>>> pair[0], pair[1]
(10, 20)
```

Define rationals:

```
>>> def rational(n, d):
        return [n, d]
>>> def numer(x):
        return x[0]
>>> def denom(x):
        return x[1]
```

Usage:

```
>>> half = rational(1, 2)
>>> third = rational(1, 3)
>>> print_rational(half)
1 / 2
>>> print_rational(mul_rationals(half, third))
1 / 6
```

Simplify fractions:

```
>>> from fractions import gcd
>>> def rational(n, d):
        g = gcd(n, d)
        return (n//g, d//g)
>>> print_rational(add_rationals(third, third))
2 / 3
```

2.3 Abstraction Barriers

Layers:

Level	Treat rationals as	Use only
Computation	Whole values	add, mul, equal, print
Arithmetic impl.	n/d parts	rational, numer, denom
Representation	Lists	list literals, indexing

Definition at the correct layer of abstraction:

```
>>> def square_rational(x):
...     return mul_rationals(x, x)
```

Violations ("breaking" once/twice the abstraction barriers):

```
>>> def square_rational_violating_once(x):
...     return rational(numer(x)*numer(x), denom(x)*denom(x))

>>> def square_rational_violating_twice(x):
...     return [x[0]*x[0], x[1]*x[1]]
```

Only the first respects abstraction barriers.

2.4 The Properties of Data

Concept:

- A data type is defined by:
  - constructors + selectors,
  - behavior conditions they satisfy.
- Any implementation meeting those conditions is valid.

Functional pair representation:

```
>>> def pair(x, y):
...     def get(index):
...         if index == 0:
...             return x
...         elif index == 1:
...             return y
...     return get

>>> def select(p, i):
        return p(i)
```

Use:

```
>>> p = pair(20, 14)
>>> select(p, 0)
20
>>> select(p, 1)
14
```

Takeaway:

- Data abstraction = interface + behavior, not representation.
- Functions alone can represent compound data.

3 Sequences

Core abstraction: an *ordered* collection with:

- **Length:** finite; empty sequence has length 0.
- **Element selection:** 0-based indexing up to (but not including) length.
- Many Python types satisfy this abstraction (e.g., lists, ranges, strings).

3.1 Lists

Lists = variable-length sequences with literals `[ ... ]`, indexing, `len`, concatenation `+`, and repetition `*`. They can be nested.

```
>>> digits = [1, 8, 2, 8]
>>> len(digits)
4
>>> digits[3]
8
>>> [2, 7] + digits * 2
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]
>>> pairs = [[10, 20], [30, 40]]
>>> pairs[1]
[30, 40]
>>> pairs[1][0]
30
```

3.2 Sequence Iteration

Iterate elements directly with `for`; can replace index-based loops. **Unpacking** binds multiple names per element. **Ranges** represent integer sequences, often used in `for`.

```
>>> def count(s, value):
...     """Count the number of occurrences of value in sequence s."""
...     total, index = 0, 0
...     while index < len(s):
...         if s[index] == value:
...             total = total + 1
...         index = index + 1
...     return total
>>> count(digits, 8)
2
>>> def count(s, value):
...     """Count the number of occurrences of value in sequence s."""
...     total = 0
...     for elem in s:
...         if elem == value:
...             total = total + 1
...     return total
>>> count(digits, 8)
2
```

For syntax & evaluation:

```
for <name> in <expression>:
    <suite>
```

Evaluate `<expression>` to an iterable, then bind `<name>` to each element (in order) and execute `<suite>`; after the loop, `<name>` is bound to the last element.

Sequence unpacking in for-headers:

```
>>> pairs = [[1, 2], [2, 2], [2, 3], [4, 4]]
>>> same_count = 0
>>> for x, y in pairs:
...     if x == y:
...         same_count = same_count + 1
>>> same_count
2
```

Ranges:

```
>>> range(1, 10) # Includes 1, but not 10
range(1, 10)
>>> list(range(5, 8))
[5, 6, 7]
>>> list(range(4))
[0, 1, 2, 3]
>>> for _ in range(3):
        print('Go Bears!')

Go Bears!
Go Bears!
Go Bears!
```

3.3 Sequence Processing

Patterns: (i) map/transform each element, (ii) filter/select some elements, (iii) aggregate to one value. We can chain together pipeline of such sequence processing operations.

List comprehensions (map/filter):

```
>>> odds = [1, 3, 5, 7, 9]
>>> [x+1 for x in odds]
[2, 4, 6, 8, 10]
>>> [x for x in odds if 25 % x == 0]
[1, 5]
```

Aggregation via built-ins (e.g., sum/min/max) and composition:

```
>>> def divisors(n):
         return [1] + [x for x in range(2, n) if n % x == 0]
>>> divisors(4)
[1, 2]
>>> divisors(12)
[1, 2, 3, 4, 6]
>>> [n for n in range(1, 1000) if sum(divisors(n)) == n]
[6, 28, 496]
```

Case study: min perimeter with integer sides

```
>>> def width(area, height):
        assert area % height == 0
        return area // height

>>> def perimeter(width, height):
        return 2 * width + 2 * height

>>> def minimum_perimeter(area):
        heights = divisors(area)
        perimeters = [perimeter(width(area, h), h) for h in heights]
        return min(perimeters)

>>> area = 80
>>> width(area, 5)
16
>>> perimeter(16, 5)
42
>>> perimeter(10, 8)
36
>>> minimum_perimeter(area)
36
>>> [minimum_perimeter(n) for n in range(1, 10)]
[4, 6, 8, 8, 12, 10, 16, 12, 12]
```

Higher-order function forms of the same patterns:

```
>>> def apply_to_all(map_fn, s):
        return [map_fn(x) for x in s]

>>> def keep_if(filter_fn, s):
        return [x for x in s if filter_fn(x)]
```

```
>>> def reduce(reduce_fn, s, initial):
        reduced = initial
        for x in s:
            reduced = reduce_fn(reduced, x)
        return reduced

>>> reduce(mul, [2, 4, 8], 1)
64
>>> def divisors_of(n):
        divides_n = lambda x: n % x == 0
        return [1] + keep_if(divides_n, range(2, n))

>>> divisors_of(12)
[1, 2, 3, 4, 6]
>>> from operator import add
>>> def sum_of_divisors(n):
        return reduce(add, divisors_of(n), 0)

>>> def perfect(n):
        return sum_of_divisors(n) == n

>>> keep_if(perfect, range(1, 1000))
[1, 6, 28, 496]
>>> apply_to_all = lambda map_fn, s: list(map(map_fn, s))
>>> keep_if = lambda filter_fn, s: list(filter(filter_fn, s))
>>> from functools import reduce
>>> from operator import mul
>>> def product(s):
        return reduce(mul, s)

>>> product([1, 2, 3, 4, 5])
120
```

3.4 Sequence Abstraction

Extra common behaviors for sequences:

- **Membership:** `in`/`not in`
- **Slicing:** `seq[start:end]` (end-exclusive); omitted bounds use extremes.

```
>>> digits               # Examples of membership
[1, 8, 2, 8]
>>> 2 in digits
True
>>> 1828 not in digits
True
>>> digits[0:2]          # Examples of slicing
[1, 8]
>>> digits[1:]
[8, 2, 8]
```

3.5 Strings

Strings are sequences of characters (no separate char type). Support length, selection, concatenation, repetition, substring membership, multiline literals, and coercion via `str`.

```
>>> 'I am string!'
'I am string!'
>>> "I've got an apostrophe"
"I've got an apostrophe"
>>> '您好'
'您好'
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
>>> 'Berkeley' + ', CA'
'Berkeley, CA'
>>> 'Shabu ' * 2
'Shabu Shabu '
>>> 'here' in "Where's Waldo?"
True
>>> """The Zen of Python                                   # Multiline string
... claims, Readability counts.
... Read more: import this."""
'The Zen of Python\nclaims, "Readability counts."\nRead more: import this.'
>>> str(2) + ' is an element of ' + str(digits)           # String coercion
'2 is an element of [1, 8, 2, 8]'
```

3.6 Trees

Hierarchical data via trees: each tree has a *root label* and *branches* (each a tree). **API:** `tree` (ctor), `label`, `branches`; plus validators.

```
>>> def tree(root_label, branches=[]):
...     for branch in branches:
...         assert is_tree(branch), 'branches must be trees'
...     return [root_label] + list(branches)
>>> def label(tree):
...     return tree[0]
>>> def branches(tree):
...     return tree[1:]
>>> def is_tree(tree):
...     if type(tree) != list or len(tree) < 1:
...         return False
...     for branch in branches(tree):
...         if not is_tree(branch):
...             return False
...     return True
>>> def is_leaf(tree):
...     return not branches(tree)
```

```
>>> t = tree(3, [tree(1), tree(2, [tree(1), tree(1)])])
>>> t
[3, [1], [2, [1], [1]]]
```

```
>>> label(t)
3
>>> branches(t)
[[1], [2, [1], [1]]]
>>> label(branches(t)[1])
2
>>> is_leaf(t)
False
>>> is_leaf(branches(t)[0])
True
```

Tree-recursive construction and processing: Building a *fibonacci tree*:

```
>>> def fib_tree(n):
        if n == 0 or n == 1:
            return tree(n)
        else:
            left, right = fib_tree(n-2), fib_tree(n-1)
            fib_n = label(left) + label(right)
            return tree(fib_n, [left, right])
>>> fib_tree(5)
[5, [2, [1], [1, [0], [1]]], [3, [1, [0], [1]], [2, [1], [1, [0], [1]]]]]
```

Counting the number of leaves in a tree:

```
>>> def count_leaves(tree):
      if is_leaf(tree):
          return 1
      else:
          branch_counts = [count_leaves(b) for b in branches(tree)]
          return sum(branch_counts)
>>> count_leaves(fib_tree(5))
8
```

Partition trees and printing partitions:

```
>>> def partition_tree(n, m):
        """Return a partition tree of n using parts of up to m."""
        if n == 0:
            return tree(True)
        elif n < 0 or m == 0:
            return tree(False)
        else:
            left = partition_tree(n-m, m)
            right = partition_tree(n, m-1)
            return tree(m, [left, right])
>>> partition_tree(2, 2)
[2, [True], [1, [1, [True], [False]], [False]]]
>>> def print_parts(tree, partition=[]):
        if is_leaf(tree):
            if label(tree):
                print(' + '.join(partition))
        else:
            left, right = branches(tree)
            m = str(label(tree))
            print_parts(left, partition + [m])
            print_parts(right, partition)
>>> print_parts(partition_tree(6, 4))
```

```
4 + 2
4 + 1 + 1
3 + 3
3 + 2 + 1
3 + 1 + 1 + 1
2 + 2 + 2
2 + 2 + 1 + 1
2 + 1 + 1 + 1 + 1
1 + 1 + 1 + 1 + 1 + 1
```

Binarization via slicing/grouping:

```
>>> def right_binarize(tree):
        """Construct a right-branching binary tree."""
        if is_leaf(tree):
            return tree
        if len(tree) > 2:
            tree = [tree[0], tree[1:]]
        return [right_binarize(b) for b in tree]
>>> right_binarize([1, 2, 3, 4, 5, 6, 7])
[1, [2, [3, [4, [5, [6, 7]]]]]]
```

3.7 Linked Lists

User-defined sequence rep using nested pairs; `'empty'` denotes the empty list. **ADT:** `link` (ctor), `first`, `rest`; `is_link` validator.

```
>>> empty = 'empty'
>>> def is_link(s):
        """s is a linked list if it is empty or a (first, rest) pair."""
        return s == empty or (len(s) == 2 and is_link(s[1]))
>>> def link(first, rest):
        """Construct a linked list from its first element and the rest."""
        assert is_link(rest), "rest must be a linked list."
        return [first, rest]
>>> def first(s):
        """Return the first element of a linked list s."""
        assert is_link(s), "first only applies to linked lists."
        assert s != empty, "empty linked list has no first element."
        return s[0]
>>> def rest(s):
        """Return the rest of the elements of a linked list s."""
        assert is_link(s), "rest only applies to linked lists."
        assert s != empty, "empty linked list has no rest."
        return s[1]
```

```
>>> four = link(1, link(2, link(3, link(4, empty))))
>>> first(four)
1
>>> rest(four)
[2, [3, [4, 'empty']]]
```

Sequence behaviors for linked lists:

```
>>> def len_link(s):
        """Return the length of linked list s."""
        length = 0
        while s != empty:
            s, length = rest(s), length + 1
        return length
>>> def getitem_link(s, i):
        """Return the element at index i of linked list s."""
        while i > 0:
            s, i = rest(s), i - 1
        return first(s)

>>> len_link(four)
4
>>> getitem_link(four, 1)
2
```

Recursive forms and utilities:

```
>>> def len_link_recursive(s):
        """Return the length of a linked list s."""
        if s == empty:
            return 0
        return 1 + len_link_recursive(rest(s))
>>> def getitem_link_recursive(s, i):
        """Return the element at index i of linked list s."""
        if i == 0:
            return first(s)
        return getitem_link_recursive(rest(s), i - 1)
>>> len_link_recursive(four)
4
>>> getitem_link_recursive(four, 1)
2
>>> def extend_link(s, t):
        """Return a list with the elements of s followed by those of t."""
        assert is_link(s) and is_link(t)
        if s == empty:
            return t
        else:
            return link(first(s), extend_link(rest(s), t))
>>> extend_link(four, four)
[1, [2, [3, [4, [1, [2, [3, [4, 'empty']]]]]]]]
>>> def apply_to_all_link(f, s):
        """Apply f to each element of s."""
        assert is_link(s)
        if s == empty:
            return s
        else:
            return link(f(first(s)), apply_to_all_link(f, rest(s)))
>>> apply_to_all_link(lambda x: x*x, four)
[1, [4, [9, [16, 'empty']]]]
>>> def keep_if_link(f, s):
        """Return a list with elements of s for which f(e) is true."""
        assert is_link(s)
        if s == empty:
            return s
        else:
            kept = keep_if_link(f, rest(s))
            if f(first(s)):
```

```
            return link(first(s), kept)
        else:
            return kept
>>> keep_if_link(lambda x: x%2 == 0, four)
[2, [4, 'empty']]
>>> def join_link(s, separator):
        """Return a string of all elements in s separated by separator."""
        if s == empty:
            return ""
        elif rest(s) == empty:
            return str(first(s))
        else:
            return str(first(s)) + separator + join_link(rest(s), separator)
>>> join_link(four, ", ")
'1, 2, 3, 4'
```

Recursive construction: partitions via linked lists

```
>>> def partitions(n, m):
...     """Return a linked list of partitions of n using parts of up to m.
...     Each partition is represented as a linked list.
...     """
...     if n == 0:
...         return link(empty, empty) # A list containing the empty partition
...     elif n < 0 or m == 0:
...         return empty
...     else:
...         using_m = partitions(n-m, m)
...         with_m = apply_to_all_link(lambda s: link(m, s), using_m)
...         without_m = partitions(n, m-1)
...         return extend_link(with_m, without_m)

>>> def print_partitions(n, m):
...     lists = partitions(n, m)
...     strings = apply_to_all_link(lambda s: join_link(s, " + "), lists)
...     print(join_link(strings, "\n"))
>>> print_partitions(6, 4)
4 + 2
4 + 1 + 1
3 + 3
3 + 2 + 1
3 + 1 + 1 + 1
2 + 2 + 2
2 + 2 + 1 + 1
2 + 1 + 1 + 1 + 1
1 + 1 + 1 + 1 + 1 + 1
```

4 Mutable Data (Summary)

Core idea: Mutable data allows programs to model entities that change over time. This enables modularity and stateful abstractions (object-oriented design).

4.1 The Object Metaphor

- Objects = data + behavior; classes create instances.

- Attributes accessed with dot notation; methods are function-valued attributes.
- All Python values are objects (numbers, strings, lists, etc.).

```
>>> from datetime import date
>>> tues = date(2014, 5, 13)
>>> print(date(2014, 5, 19) - tues)
6 days, 0:00:00
>>> tues.year
2014
>>> tues.strftime('%A, %B %d')
'Tuesday, May 13'
>>> '1234'.isnumeric()
True
>>> 'rOBERT dE nIRO'.swapcase()
'Robert De Niro'
>>> 'eyes'.upper().endswith('YES')
True
```

4.2 2.4.2 Sequence Objects

- Lists are mutable (can change in place).
- Methods: `pop`, `remove`, `append`, `extend`, slice assignment.
- Aliasing: multiple names can refer to the same list.
- `is` tests identity; `==` tests equality of contents.

```
>>> chinese = ['coin', 'string', 'myriad']
>>> suits = chinese
>>> suits.pop()
'myriad'
>>> suits.remove('string')
>>> suits.append('cup')
>>> suits.extend(['sword', 'club'])
>>> suits[2] = 'spade'
>>> suits
['coin', 'cup', 'spade', 'club']
>>> suits[0:2] = ['heart', 'diamond']
>>> suits
['heart', 'diamond', 'spade', 'club']
>>> chinese
['heart', 'diamond', 'spade', 'club']
```

```
>>> nest = list(suits)
>>> nest[0] = suits
>>> suits.insert(2, 'Joker')
>>> nest
[['heart', 'diamond', 'Joker', 'spade', 'club'], 'diamond', 'spade', 'club']
>>> nest[0].pop(2)
'Joker'
>>> suits
['heart', 'diamond', 'spade', 'club']
>>> suits is nest[0]
True
>>> suits == ['heart', 'diamond', 'spade', 'club']
True
```

```
>>> from unicodedata import lookup
>>> [lookup('WHITE ' + s.upper() + ' SUIT') for s in suits]
['♡', '♢', '♤', '♧']
```

4.3 Tuples

- Immutable sequences; parentheses optional.
- Support length, indexing, `count`, `index`.
- Contained mutable elements can still change.

```
>>> 1, 2 + 3
(1, 5)
>>> ("the", 1, ("and", "only"))
('the', 1, ('and', 'only'))
>>> type((10, 20))
<class 'tuple'>
>>> ()
()
>>> (10,)
(10,)
>>> code = ("up", "up", "down", "down") + ("left", "right") * 2
>>> len(code)
8
>>> code[3]
'down'
>>> code.count("down")
2
>>> code.index("left")
4
>>> nest = (10, 20, [30, 40])
>>> nest[2].pop()
40
```

4.4 Dictionaries

- Key – value mappings; keys must be immutable.
- Unordered; lookup and assignment via `[key]`.
- `get(key, default)` avoids errors on missing keys.
- Dict comprehensions create new dicts.

```
>>> numerals = 'I': 1.0, 'V': 5, 'X': 10
>>> numerals['X']
10
>>> numerals['I'] = 1
>>> numerals['L'] = 50
>>> numerals
'I': 1, 'X': 10, 'L': 50, 'V': 5
>>> sum(numerals.values())
66
>>> dict([(3, 9), (4, 16), (5, 25)])
```

```
3: 9, 4: 16, 5: 25
>>> numerals.get('A', 0)
0
>>> numerals.get('V', 0)
5
>>> x: x*x for x in range(3,6)
3: 9, 4: 16, 5: 25
```

4.5 Local State & `nonlocal`

- Functions can have evolving local state with `nonlocal`.
- Enables persistent state across calls.

```
>>> def make_withdraw(balance):
...     """Return a withdraw function that draws down balance."""
...     def withdraw(amount):
...         nonlocal balance
...         if amount > balance:
...             return 'Insufficient funds'
...         balance = balance - amount
...         return balance
...     return withdraw
>>> withdraw = make_withdraw(100)
>>> withdraw(25)
75
>>> withdraw(25)
50
>>> withdraw(60)
'Insufficient funds'
```

5 Object-Oriented Programming (Summary)

Essence: OOP unifies abstraction, state, and message passing. Objects encapsulate data + behavior; classes define object templates and promote modular, extensible design.

5.1 Objects and Classes

- Objects combine local state + behavior; defined by their class.
- Instance attributes belong to each object; class attributes are shared.
- Methods operate on the object' s own data (`self`).

```
>>> a = Account('Kirk')
>>> a.holder
'Kirk'
>>> a.balance
0
>>> a.deposit(15)
15
>>> a.withdraw(10)
5
```

```
>>> a.balance
5
>>> a.withdraw(10)
'Insufficient funds'
```

5.2 Defining Classes

- `class` statements define new types.
- `__init__` initializes new instances; `self` refers to the instance.
- Each instance has independent state.

```
>>> class Account:
...     def __init__(self, account_holder):
...         self.balance = 0
...         self.holder = account_holder
```

```
>>> a = Account('Kirk')
>>> a.balance
0
>>> b = Account('Spock')
>>> b.balance = 200
>>> [acc.balance for acc in (a, b)]
[0, 200]
>>> a is not b
True
```

```
>>> class Account:
...     def __init__(self, account_holder):
...         self.balance = 0
...         self.holder = account_holder
...     def deposit(self, amount):
...         self.balance = self.balance + amount
...         return self.balance
...     def withdraw(self, amount):
...         if amount > self.balance:
...             return 'Insufficient funds'
...         self.balance = self.balance - amount
...         return self.balance
```

```
>>> spock_account = Account('Spock')
>>> spock_account.deposit(100)
100
>>> spock_account.withdraw(90)
10
>>> spock_account.withdraw(90)
'Insufficient funds'
>>> spock_account.holder
'Spock'
```

5.3 Message Passing and Dot Expressions

- Dot notation formalizes message passing.
- `<expr>.<name>` fetches an attribute or bound method.
- `getattr`, `hasattr` provide dynamic access.
- Methods differ from functions: bound to object automatically.

```
>>> getattr(spock_account, 'balance')
10
>>> hasattr(spock_account, 'deposit')
True
>>> type(Account.deposit)
<class 'function'>
>>> type(spock_account.deposit)
<class 'method'>
>>> Account.deposit(spock_account, 1001)
1011
>>> spock_account.deposit(1000)
2011
```

5.4 Class Attributes

- Class attributes shared by all instances.
- Instance attributes shadow class ones.
- Assigning to instance creates a local override.

```
>>> class Account:
...     interest = 0.02
...     def __init__(self, account_holder):
...         self.balance = 0
...         self.holder = account_holder
>>> spock_account = Account('Spock')
>>> kirk_account = Account('Kirk')
>>> spock_account.interest
0.02
>>> Account.interest = 0.04
>>> spock_account.interest
0.04
>>> kirk_account.interest
0.04
>>> kirk_account.interest = 0.08
>>> spock_account.interest
0.04
>>> Account.interest = 0.05
>>> spock_account.interest
0.05
>>> kirk_account.interest
0.08
```

5.5 Inheritance

- Subclasses extend or override base class behavior.
- Support “is-a” relationships; code reuse via inheritance.

```
>>> class Account:
...     """A bank account that has a non-negative balance."""
...     interest = 0.02
...     def __init__(self, account_holder):
...         self.balance = 0
...         self.holder = account_holder
...     def deposit(self, amount):
...         self.balance += amount
...         return self.balance
...     def withdraw(self, amount):
...         if amount > self.balance:
...             return 'Insufficient funds'
...         self.balance -= amount
...         return self.balance
```

```
>>> class CheckingAccount(Account):
...     """A bank account that charges for withdrawals."""
...     withdraw_charge = 1
...     interest = 0.01
...     def withdraw(self, amount):
...         return Account.withdraw(self, amount + self.withdraw_charge)
```

```
>>> checking = CheckingAccount('Sam')
>>> checking.deposit(10)
10
>>> checking.withdraw(5)
4
>>> checking.interest
0.01
```

5.6 Using Inheritance

- Name lookup searches instance → class → base classes (MRO).
- `self` ensures correct dispatch when calling parent methods.
- Interfaces = shared attribute names and behaviors across types.

```
>>> def deposit_all(winners, amount=5):
...     for account in winners:
...         account.deposit(amount)
```

5.7 Multiple Inheritance

- Classes can inherit from multiple bases. This can lead to ambiguity: in what order do we search the superclasses to resolve a name?

  In a diamond shape, Python resolves names from left to right, then upwards (see code below, after `AsSeenOnTVAccount.mro()`).

- Further reading: Name resolution uses MRO (C3 algorithm) – out of scope of this module.

```
>>> class SavingsAccount(Account):
...     deposit_charge = 2
...     def deposit(self, amount):
...         return Account.deposit(self, amount - self.deposit_charge)
>>> class AsSeenOnTVAccount(CheckingAccount, SavingsAccount):
...     def __init__(self, account_holder):
...         self.holder = account_holder
...         self.balance = 1
>>> such_a_deal = AsSeenOnTVAccount("John")
>>> such_a_deal.balance
1
>>> such_a_deal.deposit(20)
19
>>> such_a_deal.withdraw(5)
13
>>> [c.__name__ for c in AsSeenOnTVAccount.mro()]
['AsSeenOnTVAccount', 'CheckingAccount', 'SavingsAccount', 'Account', 'object']
```

5.8 The Role of Objects

- OOP promotes modularity, encapsulation, abstraction barriers.
- Classes define logic and state; objects manage internal data.
- OOP and functional styles complement each other.

6 Implementing Classes and Objects (Summary)

Essence: We can build an object system from scratch using only functions and dictionaries. Instances and classes are dispatch dictionaries that respond to messages like `'get'`, `'set'`, and `'new'`. This mirrors Python' s object system conceptually (attributes, methods, inheritance).

6.1 Instances

- An instance stores its attributes in a local dictionary.
- `'get'` looks up attributes locally, then in the class.
- If a function is found in the class, it is bound to the instance.

```
>>> def make_instance(cls):
...     """Return a new object instance, which is a dispatch dictionary."""
...     def get_value(name):
...         if name in attributes:
```

```
...             return attributes[name]
...         else:
...             value = cls['get'](name)
...             return bind_method(value, instance)
...     def set_value(name, value):
...         attributes[name] = value
...     attributes =
...     instance = 'get': get_value, 'set': set_value
...     return instance
```

Bound methods: create a callable that inserts `self` automatically.

```
>>> def bind_method(value, instance):
...     """Return a bound method if value is callable, or value otherwise."""
...     if callable(value):
...         def method(*args):
...             return value(instance, *args)
...         return method
...     else:
...         return value
```

6.2 Classes

- A class is also a dispatch dictionary.
- Supports `'get'`, `'set'`, and `'new'`.
- Looks up names in its own attributes or its base class.

```
>>> def make_class(attributes, base_class=None):
...     """Return a new class, which is a dispatch dictionary."""
...     def get_value(name):
...         if name in attributes:
...             return attributes[name]
...         elif base_class is not None:
...             return base_class['get'](name)
...     def set_value(name, value):
...         attributes[name] = value
...     def new(*args):
...         return init_instance(cls, *args)
...     cls = 'get': get_value, 'set': set_value, 'new': new
...     return cls
```

```
>>> def init_instance(cls, *args):
...     """Return a new object with type cls, initialized with args."""
...     instance = make_instance(cls)
...     init = cls['get']('__init__')
...     if init:
...         init(instance, *args)
...     return instance
```

6.3 Using Implemented Objects

Defining the Account class:

```
>>> def make_account_class():
...     """Return the Account class, which has deposit and withdraw methods."""
...     interest = 0.02
...     def __init__(self, account_holder):
...         self['set']('holder', account_holder)
...         self['set']('balance', 0)
...     def deposit(self, amount):
...         """Increase the account balance by amount and return the new balance."""
...         new_balance = self['get']('balance') + amount
...         self['set']('balance', new_balance)
...         return self['get']('balance')
...     def withdraw(self, amount):
...         """Decrease the account balance by amount and return the new balance."""
...         balance = self['get']('balance')
...         if amount > balance:
...             return 'Insufficient funds'
...         self['set']('balance', balance - amount)
...         return self['get']('balance')
...     return make_class(locals())
```

Creating and using instances:

```
>>> Account = make_account_class()
>>> kirk_account = Account['new']('Kirk')
>>> kirk_account['get']('holder')
'Kirk'
>>> kirk_account['get']('interest')
0.02
>>> kirk_account['get']('deposit')(20)
20
>>> kirk_account['get']('withdraw')(5)
15
>>> kirk_account['set']('interest', 0.04)
>>> Account['get']('interest')
0.02
```

Inheritance (CheckingAccount subclass):

```
>>> def make_checking_account_class():
...     """Return the CheckingAccount class, which imposes a $1 withdrawal fee."""
...     interest = 0.01
...     withdraw_fee = 1
...     def withdraw(self, amount):
...         fee = self['get']('withdraw_fee')
...         return Account['get']('withdraw')(self, amount + fee)
...     return make_class(locals(), Account)
```

```
>>> CheckingAccount = make_checking_account_class()
>>> jack_acct = CheckingAccount['new']('Spock')
>>> jack_acct['get']('interest')
0.01
```

```
>>> jack_acct['get']('deposit')(20)
20
>>> jack_acct['get']('withdraw')(5)
14
```

Conclusion: This hand-built object system behaves similarly to Python' s native one. Each instance has a local attributes dictionary (like `__dict__`), and classes define shared behavior and constructors.

7 Object Abstraction (Summary)

Essence: The object system enables data abstraction, supporting multiple representations and shared behavior. A **generic function** operates on values of different types—implemented through shared interfaces, type dispatching, or type coercion.

7.1 String Conversion

- Objects should provide both a human-readable (`str`) and an evaluable (`repr`) string.
- `repr(object)` calls `object.__repr__()`, and `str(object)` calls `object.__str__()`.
- Users can define these methods in their own classes.

```
>>> 12e12
12000000000000.0
>>> print(repr(12e12))
12000000000000.0
>>> from datetime import date
>>> tues = date(2011, 9, 12)
>>> repr(tues)
'datetime.date(2011, 9, 12)'
>>> str(tues)
'2011-09-12'
>>> tues.__repr__()
'datetime.date(2011, 9, 12)'
>>> tues.__str__()
'2011-09-12'
```

7.2 Special Methods

- `__bool__`, `__len__`, `__getitem__`, `__call__`, `__add__`, etc. define how objects behave with Python' s built-in operations.
- Define these to make user objects behave like built-ins.

Boolean values:

```
>>> Account.__bool__ = lambda self: self.balance != 0
>>> bool(Account('Jack'))
False
>>> if not Account('Jack'):
...     print('Jack has nothing')
Jack has nothing
```

Sequences:

```
>>> len('Go Bears!')
9
>>> 'Go Bears!'.__len__()
9
>>> 'Go Bears!'[3]
'B'
>>> 'Go Bears!'.__getitem__(3)
'B'
```

Callable objects:

```
>>> def make_adder(n):
...     def adder(k):
...         return n + k
...     return adder
>>> add_three = make_adder(3)
>>> add_three(4)
7
>>> class Adder(object):
...     def __init__(self, n):
...         self.n = n
...     def __call__(self, k):
...         return self.n + k
>>> add_three_obj = Adder(3)
>>> add_three_obj(4)
7
```

7.3 Multiple Representations

- Complex numbers can be represented in rectangular (`real, imag`) or polar (`magnitude, angle`) form.
- Both share an interface: `real`, `imag`, `magnitude`, `angle`.
- Use `@property` to compute dependent attributes.

```
>>> class Number:
...     def __add__(self, other):
...         return self.add(other)
...     def __mul__(self, other):
...         return self.mul(other)
```

```
>>> from math import atan2
>>> class ComplexRI(Number):
...     def __init__(self, real, imag):
...         self.real, self.imag = real, imag
...     @property
...     def magnitude(self):
...         return (self.real ** 2 + self.imag ** 2) ** 0.5
...     @property
...     def angle(self):
...         return atan2(self.imag, self.real)
...     def __repr__(self):
...         return 'ComplexRI(0:g, 1:g)'.format(self.real, self.imag)
```

```
>>> from math import sin, cos, pi
>>> class ComplexMA(Number):
...     def __init__(self, magnitude, angle):
...         self.magnitude, self.angle = magnitude, angle
...     @property
...     def real(self):
...         return self.magnitude * cos(self.angle)
...     @property
...     def imag(self):
...         return self.magnitude * sin(self.angle)
...     def __repr__(self):
...         return 'ComplexMA(0:g, 1:g * pi)'.format(self.magnitude, self.angle/pi)
```

```
>>> ComplexRI(1, 2) + ComplexMA(2, pi/2)
ComplexRI(1, 4)
>>> ComplexRI(0, 1) * ComplexRI(0, 1)
ComplexMA(1, 1 * pi)
```

7.4 Generic Functions

Rational numbers:

```
>>> from fractions import gcd
>>> class Rational(Number):
...     def __init__(self, numer, denom):
...         g = gcd(numer, denom)
...         self.numer, self.denom = numer // g, denom // g
...     def __repr__(self):
...         return 'Rational(0, 1)'.format(self.numer, self.denom)
...     def add(self, other):
...         nx, dx, ny, dy = self.numer, self.denom, other.numer, other.denom
...         return Rational(nx * dy + ny * dx, dx * dy)
...     def mul(self, other):
...         return Rational(self.numer * other.numer, self.denom * other.denom)
```

```
>>> Rational(2, 5) + Rational(1, 10)
Rational(1, 2)
>>> Rational(1, 4) * Rational(2, 3)
Rational(1, 6)
```

Type dispatching:

```
>>> def is_real(c):
...     if isinstance(c, ComplexRI):
...         return c.imag == 0
...     elif isinstance(c, ComplexMA):
...         return c.angle % pi == 0
```

```
>>> Rational.type_tag = 'rat'
>>> Complex.type_tag = 'com'
>>> def add_complex_and_rational(c, r):
...     return ComplexRI(c.real + r.numer/r.denom, c.imag)
```

```
>>> def mul_complex_and_rational(c, r):
...     r_mag, r_angle = r.numer/r.denom, 0
...     if r_mag < 0:
...         r_mag, r_angle = -r_mag, pi
...     return ComplexMA(c.magnitude * r_mag, c.angle + r_angle)
```

```
>>> class Number:
...     def __add__(self, other):
...         if self.type_tag == other.type_tag:
...             return self.add(other)
...         elif (self.type_tag, other.type_tag) in self.adders:
...             return self.cross_apply(other, self.adders)
...     def __mul__(self, other):
...         if self.type_tag == other.type_tag:
...             return self.mul(other)
...         elif (self.type_tag, other.type_tag) in self.multipliers:
...             return self.cross_apply(other, self.multipliers)
...     def cross_apply(self, other, cross_fns):
...         cross_fn = cross_fns[(self.type_tag, other.type_tag)]
...         return cross_fn(self, other)
...     adders = ("com", "rat"): add_complex_and_rational,
...               ("rat", "com"): add_complex_and_rational
...     multipliers = ("com", "rat"): mul_complex_and_rational,
...                    ("rat", "com"): mul_complex_and_rational
```

```
>>> ComplexRI(1.5, 0) + Rational(3, 2)
ComplexRI(3, 0)
>>> Rational(-1, 2) * ComplexMA(4, pi/2)
ComplexMA(2, 1.5 * pi)
```

Coercion:

```
>>> def rational_to_complex(r):
...     return ComplexRI(r.numer/r.denom, 0)
```

```
>>> class Number:
...     def __add__(self, other):
...         x, y = self.coerce(other)
...         return x.add(y)
...     def __mul__(self, other):
...         x, y = self.coerce(other)
...         return x.mul(y)
...     def coerce(self, other):
...         if self.type_tag == other.type_tag:
...             return self, other
...         elif (self.type_tag, other.type_tag) in self.coercions:
...             return (self.coerce_to(other.type_tag), other)
...         elif (other.type_tag, self.type_tag) in self.coercions:
...             return (self, other.coerce_to(self.type_tag))
...     def coerce_to(self, other_tag):
...         fn = self.coercions[(self.type_tag, other_tag)]
...         return fn(self)
...     coercions = ('rat', 'com'): rational_to_complex
```

Summary: Generic functions can be created through:

1. **Shared interfaces:** same method names, different implementations.
2. **Type dispatching:** select function by type tags.
3. **Coercion:** convert one type into another to unify operations.

8 Efficiency (Summary)

Idea. Efficiency concerns the *resources* (time, space) a program uses. We reason about growth using counts of key events (e.g., calls) and asymptotic notation (e.g., $\Theta(\cdot)$).

8.1 Measuring Efficiency

- Count *events* (e.g., calls) instead of wall-clock time.
- **Tree recursion** (e.g., Fibonacci) repeats work exponentially.

```
>>> def fib(n):
...     if n == 0:
...         return 0
...     if n == 1:
...         return 1
...     return fib(n-2) + fib(n-1)
>>> fib(5)
5
```

Counting calls:

```
>>> def count(f):
...     def counted(*args):
...         counted.call_count += 1
...         return f(*args)
...     counted.call_count = 0
...     return counted
>>> fib = count(fib)
>>> fib(19)
4181
>>> fib.call_count
13529
```

Space (max active frames):

```
>>> def count_frames(f):
...     def counted(*args):
...         counted.open_count += 1
...         counted.max_count = max(counted.max_count, counted.open_count)
...         result = f(*args)
...         counted.open_count -= 1
...         return result
...     counted.open_count = 0
...     counted.max_count = 0
...     return counted
>>> fib = count_frames(fib)
>>> fib(19)
4181
```

```
>>> fib.open_count
0
>>> fib.max_count
19
>>> fib(24)
46368
>>> fib.max_count
24
```

Takeaway. Time grows very fast (exponential); space is $\Theta(n)$ for `fib(n)` due to recursion depth.

8.2 Memoization

- Cache results to avoid repeated work in tree recursion.
- Requires immutable / hashable arguments (dict keys).

```
>>> def memo(f):
...     cache =
...     def memoized(n):
...         if n not in cache:
...             cache[n] = f(n)
...         return cache[n]
...     return memoized
>>> counted_fib = count(fib)
>>> fib = memo(counted_fib)
>>> fib(19)
4181
>>> counted_fib.call_count
20
>>> fib(34)
5702887
>>> counted_fib.call_count
35
```

8.3 Orders of Growth

- Abstract cost as a function $R(n)$ of input size n; compare by *order of growth*.
- Example: count divisors up to $\sqrt{n}$.

```
>>> from math import sqrt
>>> def count_factors(n):
...     sqrt_n = sqrt(n)
...     k, factors = 1, 0
...     while k < sqrt_n:
...         if n % k == 0:
...             factors += 2
...         k += 1
...     if k * k == n:
...         factors += 1
...     return factors
```

Time. The loop runs $\lfloor\sqrt{n}\rfloor - 1$ times; total work is $w \cdot \sqrt{n} + v$ for constants w, v, so time is $\Theta(\sqrt{n})$.

Theta notation. $R(n) = \Theta(f(n))$ if $\exists k_1, k_2, m > 0$ such that

$$k_1 f(n) \leq R(n) \leq k_2 f(n) \quad \text{for all } n \geq m.$$

8.4 Example: Exponentiation

Compute b^n.

Linear recursion/iteration ($\Theta(n)$ steps):

```
>>> def exp(b, n):
...     if n == 0:
...         return 1
...     return b * exp(b, n-1)
>>> def exp_iter(b, n):
...     result = 1
...     for _ in range(n):
...         result = result * b
...     return result
```

Successive squaring ($\Theta(\log n)$ steps, $\Theta(\log n)$ space recursively):

$$b^n = \begin{cases} \left(b^{n/2}\right)^2, & \text{if } n \text{ is even} \\ b \cdot b^{n-1}, & \text{if } n \text{ is odd} \end{cases}$$

```
>>> def square(x):
...     return x*x
>>> def fast_exp(b, n):
...     if n == 0:
...         return 1
...     if n % 2 == 0:
...         return square(fast_exp(b, n//2))
...     else:
...         return b * fast_exp(b, n-1)
>>> fast_exp(2, 100)
1267650600228229401496703205376
```

Reason. Each even step halves n; number of multiplications grows as $\Theta(\log n)$.

8.5 Growth Categories

- Constants don't change order: $\Theta(n)$ and $\Theta(500n)$ are equivalent.
- Log bases don't matter: $\log_2 n$ and $\log_{10} n$ differ by a constant.
- Nested loops multiply orders.
- Drop lower-order terms: $\Theta(n^2 + n) = \Theta(n^2)$.

Overlap example (nested):

```
>>> def overlap(a, b):
...     count = 0
...     for item in a:
...         if item in b:
...             count += 1
...     return count
>>> overlap([1, 3, 2, 2, 5, 1], [5, 4, 2])
3
```

Using list membership, `item in b` is $\Theta(n)$ (size of `b`). Repeated $\Theta(m)$ times (size of `a`) $\Rightarrow \Theta(mn)$.

Lower-order term example:

```
>>> def one_more(a):
...     return overlap([x-1 for x in a], a)
>>> one_more([3, 14, 15, 9])
1
```

List comprehension is $\Theta(n)$; `overlap` is $\Theta(n^2) \Rightarrow$ total $\Theta(n^2)$.

Common categories:

Complexity class	Math.	Intuition	Example
Constant	$\Theta(1)$	independent of input	e.g. `abs`
Logarithmic	$\Theta(\log n)$	doubles input $\Rightarrow$ +const work	`fast_exp`
Linear	$\Theta(n)$	+1 input $\Rightarrow$ +const work	`exp/exp_iter`
Quadratic	$\Theta(n^2)$	nested linear passes	`one_more`
Exponential	$\Theta(b^n)$	+1 input $\Rightarrow \times b$ work	`fib`

Fibonacci growth (exponential). The nth Fibonacci number satisfies

$$F_n = \frac{\varphi^n - \psi^n}{\sqrt{5}} \quad \text{with} \quad \varphi = \frac{1+\sqrt{5}}{2}, \;\; \psi = \frac{1-\sqrt{5}}{2},$$

and in particular F_n is the nearest integer to $\dfrac{\varphi^n}{\sqrt{5}}$. The naïve tree-recursive computation has time $\Theta(\varphi^n)$.

9 Pandas DataFrames (Quick, Practical Guide)

9.1 What is pandas?

- **pandas** is a popular open-source Python library (built atop NumPy) for fast, flexible data wrangling and analysis.
- Core containers: **Series** (1D, labeled) and **DataFrame** (2D, labeled rows & columns).
- Integrates with the PyData stack (NumPy, Matplotlib, SciPy, scikit-learn), and can scale via **pandas-on-Spark** (a.k.a. pandas API on Apache Spark) for distributed workloads.

9.2 Creating DataFrames

```
>>> import pandas as pd
>>> import numpy as np
>>> df = pd.DataFrame(
...     'name': ['Ada', 'Grace', 'Edsger'],
...      'year': [1815, 1906, 1932],
...      'score': [9.5, 9.9, 9.7]
... )
>>> df
     name  year  score
0     Ada  1815    9.5
1   Grace  1906    9.9
```

```
2  Edsger  1932    9.7

>>> # From a 2D NumPy array + labels
>>> arr = np.arange(6).reshape(3, 2)
>>> pd.DataFrame(arr, columns=['A', 'B'], index=['r0','r1','r2'])
     A  B
r0   0  1
r1   2  3
r2   4  5

>>> # From CSV / Parquet / SQL (examples)
>>> df = pd.read_csv('data.csv', parse_dates=['timestamp'])
>>> df = pd.read_parquet('data.parquet')
>>> import sqlite3; con = sqlite3.connect('db.sqlite')
>>> df = pd.read_sql('SELECT * FROM users', con)
```

9.3 Inspecting, dtypes, and memory

```
>>> df.head(2)
     name  year  score
0     Ada  1815    9.5
1   Grace  1906    9.9

>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   name    3 non-null      object
 1   year    3 non-null      int64
 2   score   3 non-null      float64
dtypes: float64(1), int64(1), object(1)
memory usage: 200.0+ bytes

>>> # Optional: use newer nullable dtypes
>>> df2 = df.astype('year': 'Int64')  # allows NA, keeps integer semantics
```

9.4 Indexing & selecting

```
>>> # Column(s)
>>> df['name']
0       Ada
1     Grace
2    Edsger
Name: name, dtype: object

>>> df[['name','score']]
     name  score
0     Ada    9.5
1   Grace    9.9
2  Edsger    9.7

>>> # Row by label/position (set index for label lookups)
>>> df2 = df.set_index('name')
>>> df2.loc['Grace']
```

```
year      1906.0
score        9.9
Name: Grace, dtype: float64

>>> df.iloc[1]  # second row
name     Grace
year      1906
score      9.9
Name: 1, dtype: object

>>> # Boolean filtering
>>> df[df['score'] >= 9.7]
     name  year  score
1   Grace  1906    9.9
2  Edsger  1932    9.7
```

9.5 Adding, dropping, renaming

```
>>> # Add / assign new columns
>>> df['decade'] = (df['year'] // 10) * 10
>>> df = df.assign(top=lambda d: d['score'] >= 9.8)
>>> df
     name  year  score  decade    top
0     Ada  1815    9.5    1810  False
1   Grace  1906    9.9    1900   True
2  Edsger  1932    9.7    1930  False

>>> # Drop columns/rows
>>> df.drop(columns=['decade'])
     name  year  score    top
0     Ada  1815    9.5  False
1   Grace  1906    9.9   True
2  Edsger  1932    9.7  False

>>> df.drop(index=[0])  # drop first row
     name  year  score    top
1   Grace  1906    9.9   True
2  Edsger  1932    9.7  False

>>> # Rename labels
>>> df.rename(columns='score':'rating')
     name  year  rating  decade    top
0     Ada  1815     9.5    1810  False
1   Grace  1906     9.9    1900   True
2  Edsger  1932     9.7    1930  False
```

9.6 Handling missing data

```
>>> s = pd.Series([1, None, 3], dtype='Int64')
>>> s.fillna(0)
0    1
1    0
2    3
dtype: Int64

>>> df_na = pd.DataFrame('A':[1, np.nan, 3], 'B':[4, 5, np.nan])
```

```
>>> df_na.dropna()    # drop rows with any NA
     A    B
0  1.0  4.0

>>> df_na.fillna('A': df_na['A'].mean(), 'B': 0)
     A    B
0  1.0  4.0
1  2.0  5.0
2  3.0  0.0
```

9.7 GroupBy, aggregation, and transforms

```
>>> g = df.groupby('decade', dropna=False)['score']
>>> g.agg(['count','mean'])
        count  mean
decade
1810        1   9.5
1900        1   9.9
1930        1   9.7

>>> # Transform keeps shape; e.g., z-score by group
>>> df.assign(z=lambda d: (d['score'] - g.transform('mean')) / g.transform('std'))
     name  year  score  decade    top   z
0     Ada  1815    9.5    1810  False NaN
1   Grace  1906    9.9    1900   True NaN
2  Edsger  1932    9.7    1930  False NaN
```

9.8 Merging, joining, and reshaping

```
>>> left  = pd.DataFrame('id':[1,2], 'city':['Paris','NYC'])
>>> right = pd.DataFrame('id':[1,1,2], 'lang':['py','js','py'])
>>> left.merge(right, on='id', how='left')
   id   city lang
0   1  Paris   py
1   1  Paris   js
2   2    NYC   py

>>> # Pivot / melt
>>> sales = pd.DataFrame(
...   'month':['Jan','Jan','Feb','Feb'],
...   'product':['A','B','A','B'],
...   'revenue':[10,20,12,18]
... )
>>> sales_p = sales.pivot(index='month', columns='product', values='revenue')
>>> sales_p
product   A   B
month
Feb      12  18
Jan      10  20

>>> sales_m = sales_p.reset_index().melt(id_vars='month', var_name='product', value_name='revenue')
>>> sales_m.sort_values(['month','product'])
  month product  revenue
0   Feb       A       12
1   Feb       B       18
2   Jan       A       10
```

```
3   Jan       B       20

>>> # Stack / unstack
>>> sales_p.stack()
month  product
Feb    A          12
       B          18
Jan    A          10
       B          20
dtype: int64
```

9.9 Dates, times, and parsing

```
>>> ts = pd.to_datetime(['2025-01-01', '01/02/2025', '2025.01.03'])
>>> ts
DatetimeIndex(['2025-01-01', '2025-01-02', '2025-01-03'], dtype='datetime64[ns]', freq=None)

>>> df_dt = pd.DataFrame('ts': ts)
>>> df_dt.assign(
...   day=df_dt['ts'].dt.day,
...   wk=df_dt['ts'].dt.isocalendar().week
... )
          ts  day   wk
0 2025-01-01    1    1
1 2025-01-02    2    1
2 2025-01-03    3    1

>>> # Robust CSV parsing
>>> pd.read_csv('events.csv', parse_dates=['timestamp'])
```

9.10 Vectorization vs iteration

```
>>> # Prefer vectorized ops over row-wise loops
>>> df['scaled'] = (df['score'] - df['score'].mean()) / df['score'].std()

>>> # If you must iterate (slow), use itertuples() over iterrows()
>>> for row in df.itertuples(index=False):
...     _ = (row.score, row.name)
```

9.11 Method chaining and pipes

```
>>> (df
...   .query('score >= 9.6')
...   .assign(rank=lambda d: d['score'].rank(ascending=False))
...   .sort_values('rank')
...   .loc[:, ['name','score','rank']]
... )
     name  score  rank
1   Grace    9.9   1.0
2  Edsger    9.7   2.0
```

9.12 Input/Output cheatsheet

```
>>> df.to_csv('out.csv', index=False)
>>> df.to_parquet('out.parquet', compression='snappy')
>>> df.to_json('out.json', orient='records', lines=True)
>>> pd.read_excel('book.xlsx', sheet_name='Sheet1')
```

9.13 Performance tips

- Use **vectorized** operations; avoid Python loops in tight paths.
- Choose efficient **dtypes** (e.g., `category` for low-cardinality strings).
- Use `.loc` column assignment (avoids SettingWithCopy pitfalls).
- For very large data: **chunked** CSV reads (`chunksize=...`) or **Parquet**; consider **pandas-on-Spark**/Dask/Polars when data outgrows RAM.

9.14 Quick "how do I⋯?"

```
>>> # Delete columns / rows / duplicates
>>> df = df.drop(columns=['decade'])
>>> df = df.drop(index=[0])
>>> df = df.drop_duplicates()

>>> # Rename
>>> df = df.rename(columns='name':'full_name')

>>> # Replace strings / values
>>> df['full_name'] = df['full_name'].str.replace('Edsger','E. Dijkstra', regex=False)

>>> # Split a string column into multiple columns
>>> s = pd.Series(['a:b', 'c:d'])
>>> s.str.split(':', expand=True)
   0  1
0  a  b
1  c  d

>>> # Apply row/column-wise (use sparingly vs vectorize)
>>> df.apply(np.sqrt, axis=0)  # columns
```

9.15 Tiny visualization (quick looks)

```
>>> ax = df.plot(kind='bar', x='name', y='score')  # Matplotlib backend
```