Iterable, Iterator, Generator

Prof. Pai H. Chou
National Tsing Hua University

Outline

- Data types that are iterable
  - any type that can be used by for-loop or *unpack
  - by supporting `__iter__` or `__getitem__`
- Iterator
  - a built-in class that makes an iterable out of any type
  - wrapped type implements `__next__` method
- Generator
  - a function that uses **`yield`** to work like an iterable
  - known as "coroutine" in other language terminology

What is an iterable?

可迭代

- a data structure that can produce one item at a time
  - sequence (`list`, `str`, `tuple`, `bytes`, etc)
  - non-sequence collection (`set`, `dict`, ...)
  - `range()`, "view" classes for dict keys and values
    - any type that can do ***`A[i]`*** (i.e., `__getitem__`)
- Typical usage
  - **`for`** *`i`* **`in`** *iterable*`:`
  - *function*`(`*`*`*iterable*`)`

Example use of iterable

- for loop over iterable
  - (sequence) list, tuple, str,
  - (unordered) set and dict

```
>>> for i in {'a', 'b', 'c'}:
...     print(i)
...
b
a
c
```

set: order not guaranteed!!!

```
>>> for i in ['a', 'b', 'c']:
...     print(i)
...
a
b
c
```

list

```
>>> for i in {'a':2, 'b':3, 'c':4}:
...     print(i)
...
a
b
c
```

dict: prints the keys, but order not guaranteed!

```
>>> for i in ('a', 'b', 'c'):
...     print(i)
...
a
b
c
```

tuple

```
>>> for i in 'abc':
...     print(i)
...
a
b
c
```

str

Example use of iterable

- range() as iterable
  - can use index also
  - has len()

```
>>> for i in range(3):
...     print(i)
...
0
1
2
```

range

```
>>> R = range(2, 20, 3)
>>> R[4]
14
>>> len(R)
6
```

can be indexed

- can be unpacked
  - assignment
  - parameter passing

```
>>> a, b, c = range(3):
>>> a
0
>>> b
1
>>> c
2
```

iterable can be unpacked

```
>>> max(*R) # max(2,5,8,11,14,17)
17
```

What makes a data type iterable?

- option 1: supports `__iter__()`
  - to return an iterator object for the iterable object
  - Called by the built-in iter() function to track state
- option 2: `__getitem__()`
  - to return the indexed item, so an iterator can select the item to produce

What is an iterator?

迭 ㄉㄧㄝˊ
代 ㄉㄞˋ
器 ㄑㄧˋ

- an object that tracks an iterable's state
  - iterable = "content", iterator = "current position"
- Explicit
  - `r = iter(iterable)` to make an iterator object
  - `next(r)` to advance the state of an iterator
- Implicit
  - made by for-loop or an unpacking assignment

Example of iterable vs. iterator

- iterable: the "content" part

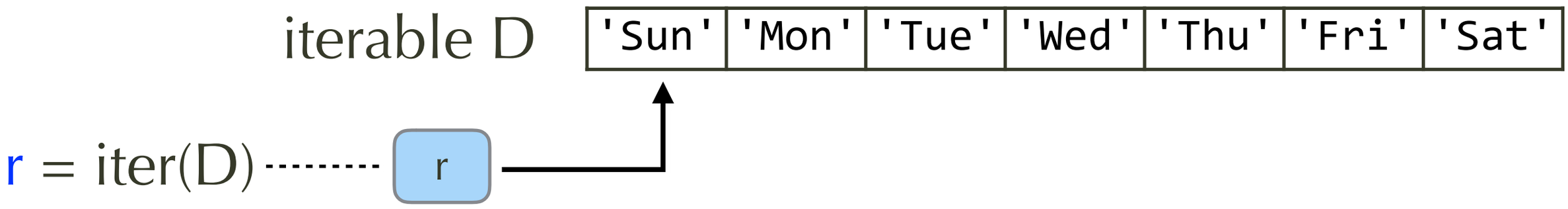
```
>>> D = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']
```

- iterator: the "position" part

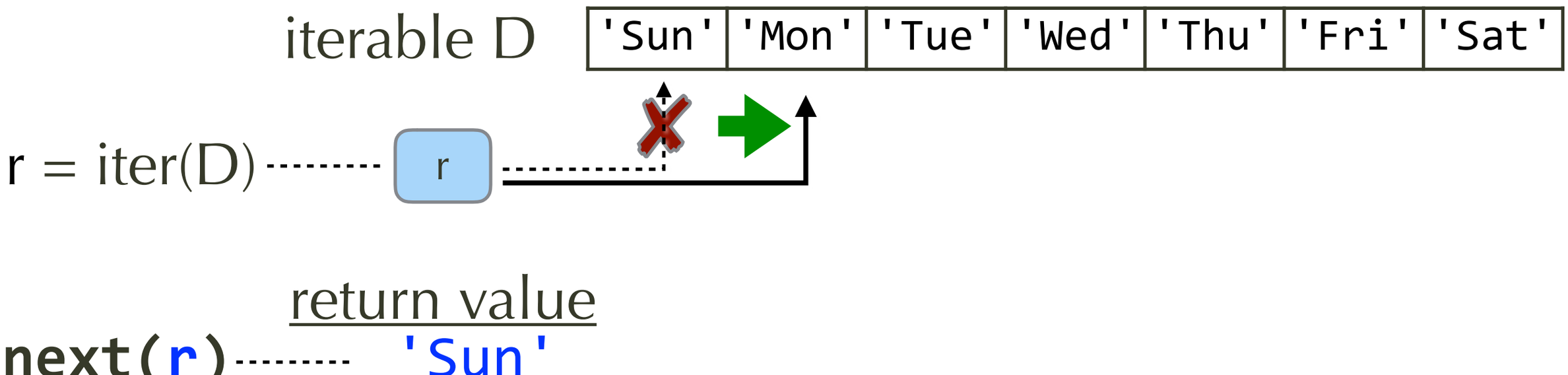
```
>>> r = iter(D)
>>> next(r)
'Sun'
>>> next(r)
'Mon'
>>> next(r)
'Tue'
>>> next(r)
'Wed'
>>> next(r)
'Thu'
```

```
>>> next(r)
'Fri'
>>> next(r)
'Sat'
>>> next(r)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

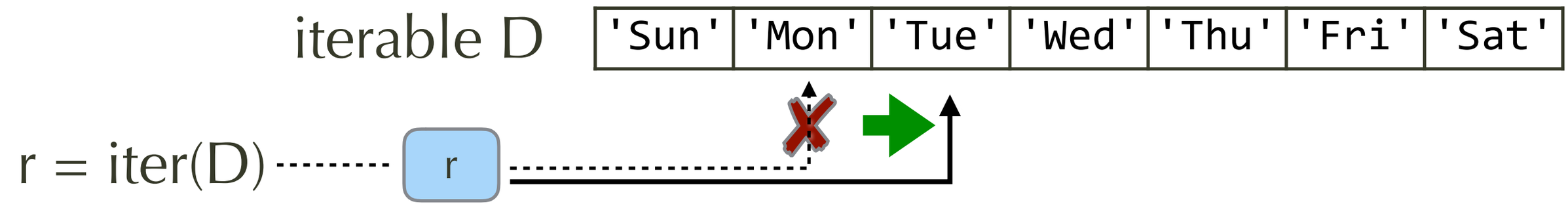
visualizing iterable vs iterator



visualizing iterable vs iterator



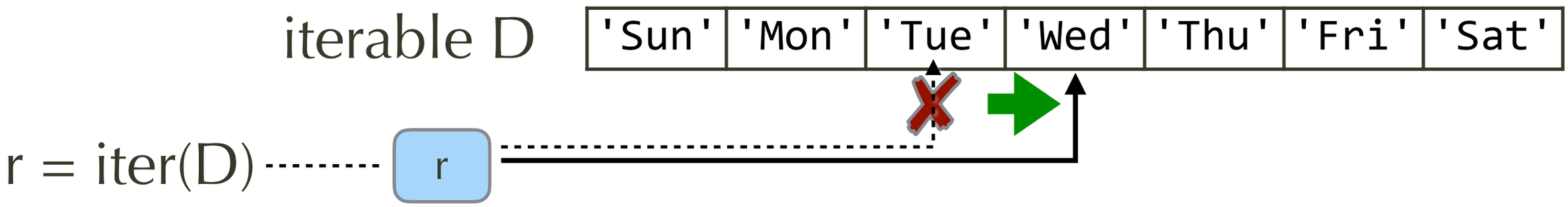
visualizing iterable vs iterator



return value

```
next(r)--------  'Sun'
next(r)--------  'Mon'
```

visualizing iterable vs iterator



return value

```
next(r)--------  'Sun'
next(r)--------  'Mon'
next(r)--------  'Tue'
```

Example: multiple iterators per iterable

- iterable: the "content" part

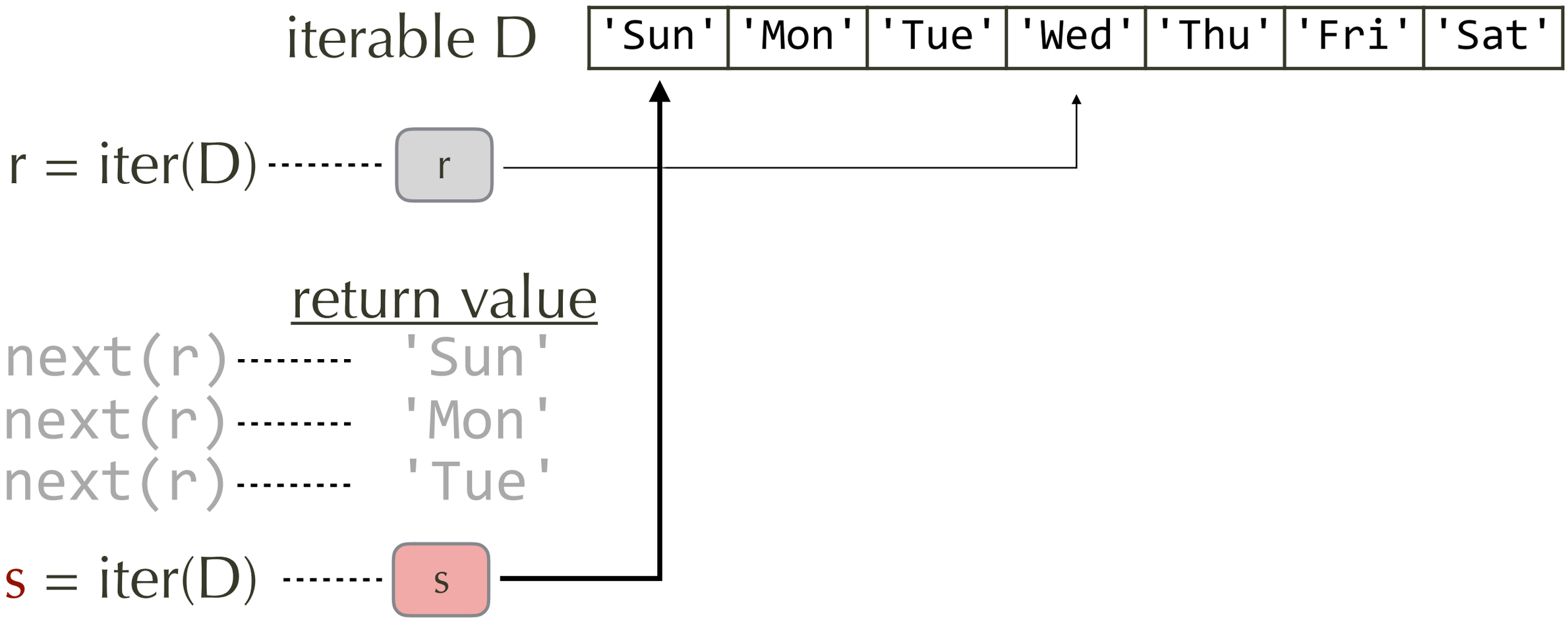
```
>>> D = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']
```

- iterator: the "position" part

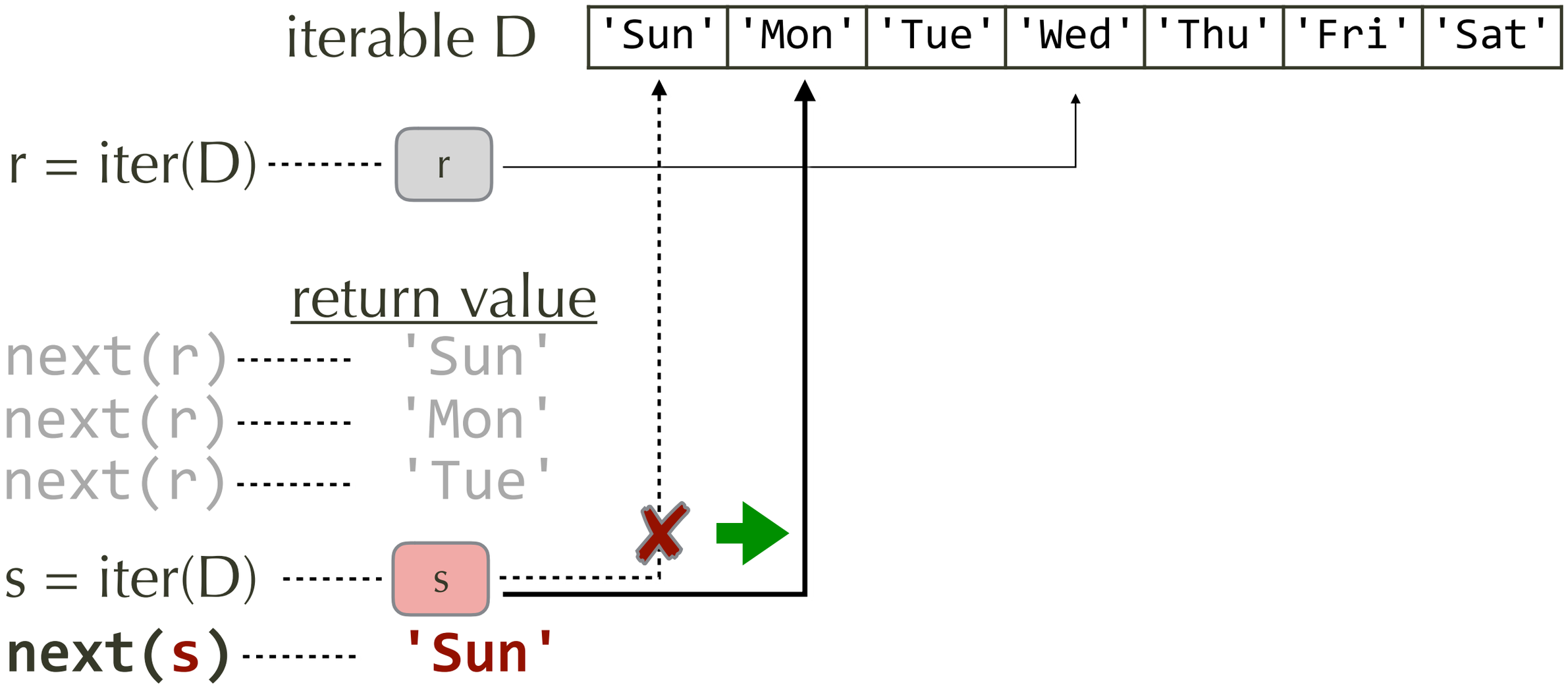
```
>>> r = iter(D)  # one iterator
>>> s = iter(D)  # another
>>> next(r)
'Sun'
>>> next(r)
'Mon'
>>> next(r)
'Tue'
>>> next(s)
'Sun'
>>> next(s)
'Mon'
```

```
>>> next(r)
'Wed'
>>> next(r)
'Thu'
>>> next(r)
'Fri'
>>> next(s)
'Tue'
>>> next(s)
'Wed'
>>> next(r)
'Sat'
```

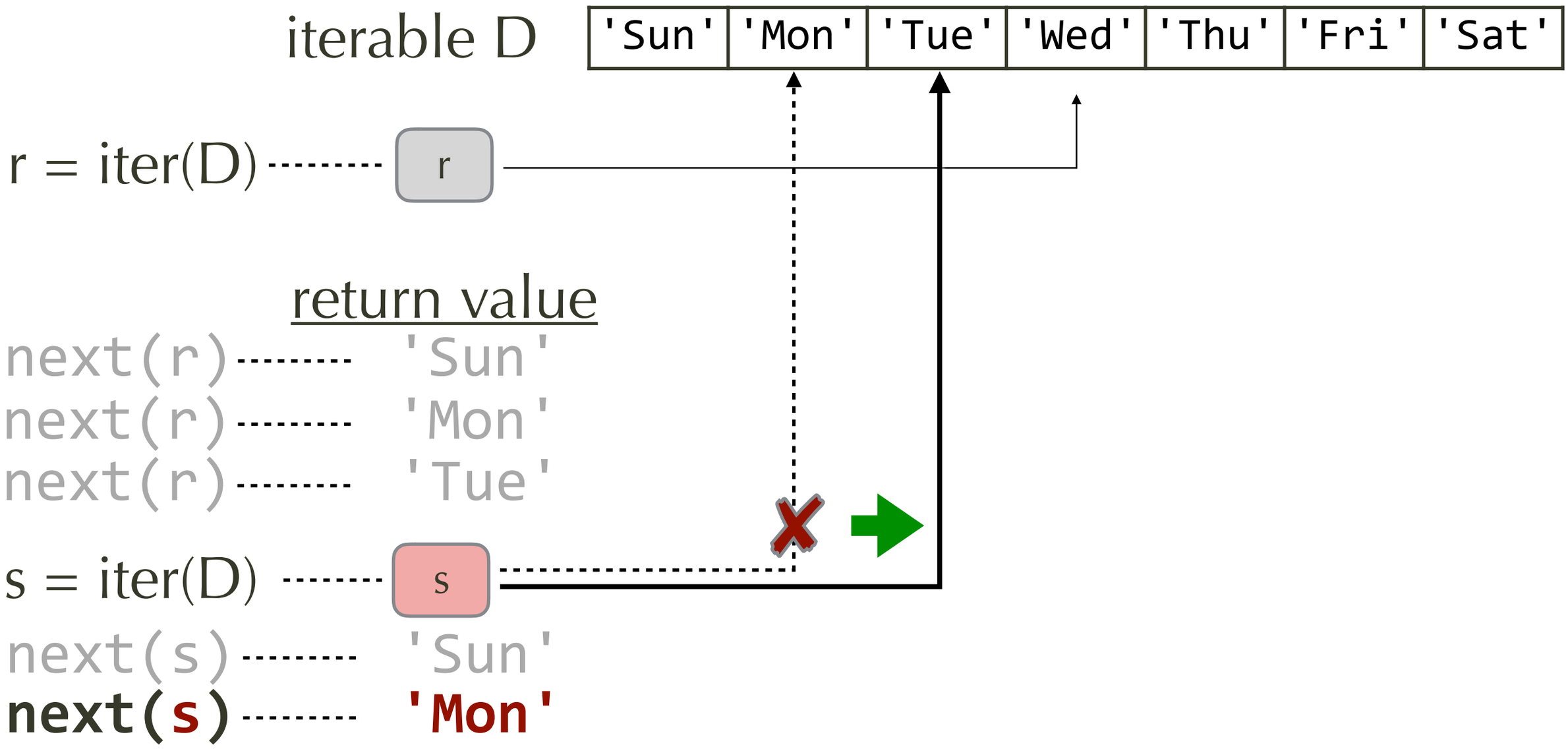
visualizing iterable vs iterator



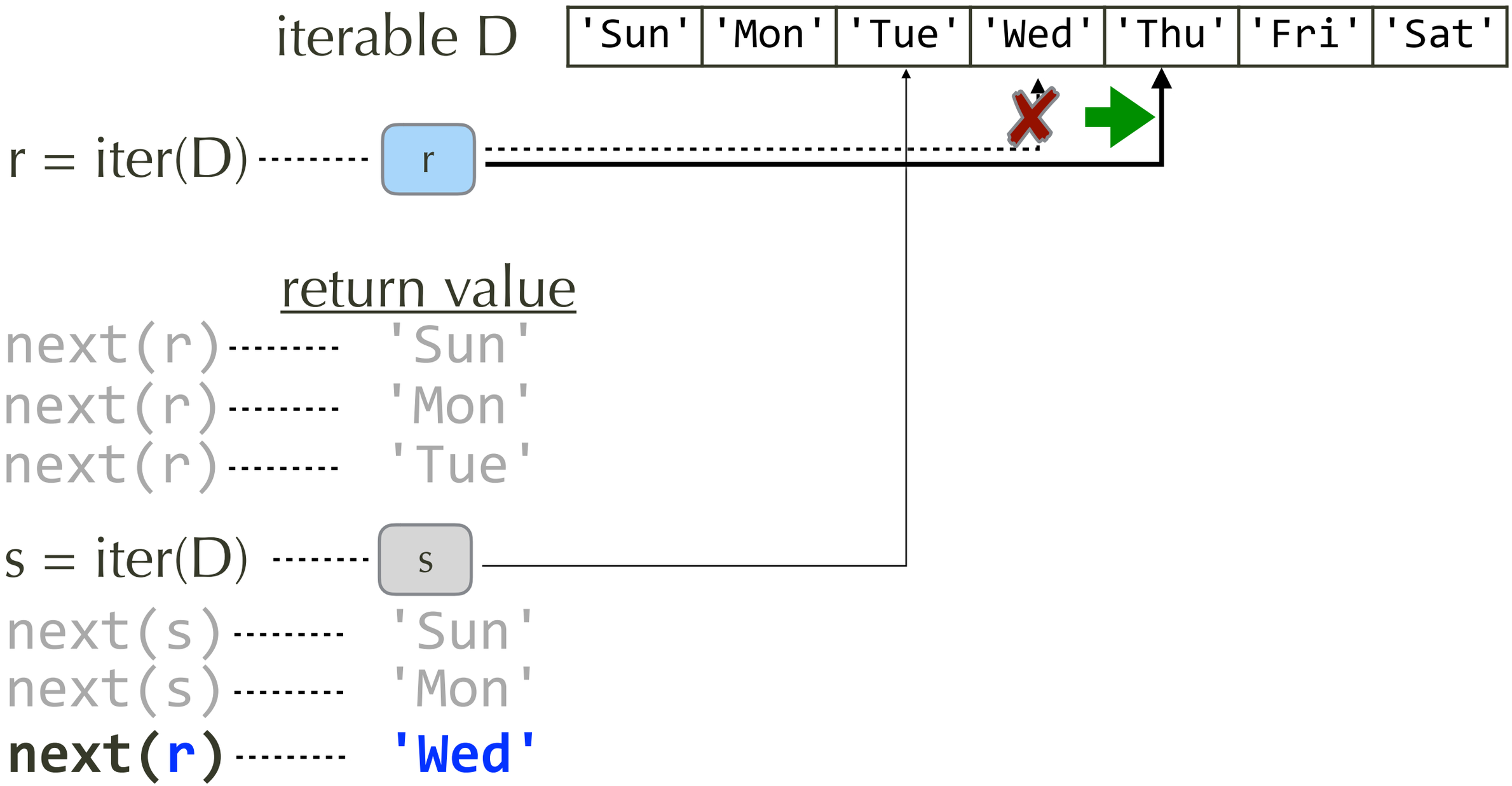
visualizing iterable vs iterator



visualizing iterable vs iterator



visualizing iterable vs iterator



how do iterators of built-in types work?

```
>>> r = iter([])
>>> type(r)
<class 'list_iterator'>
>>> type(iter({}))
<class 'dict_keyiterator'>
>>> type(iter(()))
<class 'tuple_iterator'>
>>> type(iter('hello'))
<class 'str_iterator'>
```

- built-in iterable type use own iterator class!
- For your own types
  - Option 1: let iter() call __getitem__ to make iterator
  - Option 2: define your own iterator class for your own iterable, and add __iter__ to your iterable

Example: Vector iterator
Option 1: default iterator

- Already supports `__getitem__()`

```
def Vector:
    def __getitem__(self, i):
        if type(i) == int:
            return self._v[i]
        elif type(i) == slice:
            ....
```

- call iter() to use default implementation

```
>>> r = iter(Vector(3, 5, 2, 7, 9))
>>> type(r)
<class 'iterator'>
>>> next(r)
3
>>> next(r)
5
>>> next(r)
2
```

Example: Vector iterator
Option 2: custom iterator

- Vector_Iterator
  - `__init__()`
  - `__next__()`

```
class Vector_Iterator:
  def __init__(self, vec):
    self._vec = vec
    self._i = 0
  def __next__(self):
    if self._i >= len(self._vec):
      raise StopIteration
    val = self._vec[self._i]
    self._i += 1
    return val
```

- add `__iter__` to Vector class
  - pass self to iterator's constructor

```
class Vector:
  def __iter__(self):
    return Vector_Iterator(self)
```

called by `iter()` to iterate over elements of Vector

called by `next()` to get next element

as an iterable Vector, you can

- Iterate over elements in for loop

```
>>> v = Vector(7, 1, 4, 3, 9, 6, 5)
>>> for i in v: print(i, end='')
...
7143965>>>
```

- Convert to sequence

```
>>> list(v)
[7, 1, 4, 3, 9, 6, 5]
```

- Use in unpacking assignment

```
>>> a, b, c, d, e, f, g = v
>>> a, b, c, d, e, f, g
(7, 1, 4, 3, 9, 6, 5)
```

Once you define iterator for Vector, you can

- Pass it as an iterable parameter

```
>>> v = Vector(7, 1, 4, 3, 9, 6, 5)
>>> max(v)  # take it as an iterable
9
```

- Pass with unpacking operator

```
>>> print(*v) # same print(v[0],v[1],v[2],v[3],v[4],v[5],v[6])
7 1 4 3 9 6 5
```

- Compare with printing w/out unpacking

```
>>> print(v)
Vector(7, 1, 4, 3, 9, 6, 5)
```

- All are done by it=iter(v) and next(it) behind the scenes

Application example: simple blackjack

- Dealer (computer) shuffles cards
  - for as many times as the player (user) requests
- Loop
  - Dear issues one card # it = iter(deck); c = next(it)
    - Player decides more cards or stop
- Ace = 1 point, 2 = 2 points...; J, Q, K = 10 points
- if total = 21: player wins
- elif total > 21: player loses
- elif total < 21, next card + total <= 21: player loses

Card representation

- Card
  - SUITS = { club, spade, heart, diamond } # ♣♠♥♦
  - FACES = ('A', 2, ..., 10, 'J', 'Q', 'K')
- Deck
  - list of 52 cards (4 suits, 13 faces)
  - method to shuffle cards by shuffling list items
  - __iter__ to return iterator on list of cards
    (no need to define DeckIterator class)

Card class

```
class Card:
    ACE, JACK, QUEEN, KING = 'A', 'J', 'Q', 'K'
    FACES = (ACE, 2, 3, 4, 5, 6, 7, 8, 9, 10, JACK, QUEEN, KING)
    SUITS = tuple(map(chr, (9824, 9827, 9829, 9830)))
    SPADE, CLUB, HEART, DIAMOND = SUITS # ♠ ♣ ♥ ♦

    def __init__(self, suit, face):
        self._suit = suit
        self._face = face

    def __int__(self):
        if self._face in {Card.JACK, Card.QUEEN, Card.KING}:
            return 10
        return 1 if self._face == Card.ACE else self._face

    def __str__(self):
        return self._suit + str(self._face)

    def __repr__(self):
        return __class__.__name__ + repr((self._suit, self._face))
```

Example use of Card class

```
>>> c = Card(Card.SPADE, 7)
>>> c
Card('♠', 7)
>>> print( c )
♠7
>>> int(c)
7
```

Deck class

```
class Deck:
    def __init__(self):
        self._deck = [Card(suit, face) \
                      for suit in Card.SUITS for face in Card.FACES]

    def shuffle(self):
        import random
        random.shuffle(self._deck)

    def __iter__(self):
        return iter(self._deck)
```

```
>>> d = Deck()
>>> list(map(str, d._deck))
['♠A', '♠2', '♠3', '♠4', '♠5', '♠6', '♠7', '♠8', '♠9', '♠10', '♠J',
'♠Q', '♠K', '♣A', '♣2', '♣3', '♣4', '♣5', '♣6', '♣7', '♣8', '♣9',
'♣10', '♣J', '♣Q', '♣K', '♥A', '♥2', '♥3', '♥4', '♥5', '♥6', '♥7',
'♥8', '♥9', '♥10', '♥J', '♥Q', '♥K', '♦A', '♦2', '♦3', '♦4', '♦5',
'♦6', '♦7', '♦8', '♦9', '♦10', '♦J', '♦Q', '♦K']
```

Deck class use

```
>>> d = Deck()
>>> di = iter(d)
>>> next(di)
Card('♠', 'A')
>>> next(di)
Card('♠', 2)
>>> d.shuffle()
>>> list(map(str, d._deck))
['♠5', '♠4', '♥4', '♠8', '♦K', '♣3', '♥7', '♠9', '♥5', '♥9', '♣4',
'♦Q', '♣7', '♥K', '♣5', '♣8', '♦J', '♣K', '♠6', '♥10', '♣J', '♠Q',
'♦3', '♠3', '♦6', '♥A', '♠K', '♦7', '♦A', '♠2', '♦9', '♦10', '♣A',
'♣9', '♥2', '♦5', '♦2', '♦4', '♠J', '♥J', '♥3', '♥Q', '♥8', '♥6',
'♠A', '♠7', '♠10', '♣10', '♣2', '♣6', '♣Q', '♦8']
>>> di = iter(d)
>>> next(di), next(di), next(di), next(di)
(Card('♠', 5), Card('♠', 4), Card('♥', 4), Card('♠', 8))
```

single-player BlackJack

```
def BlackJack():
    D = Deck()      # make a deck of cards
    D.shuffle()     # shuffle once here, but could loop
    total = 0       # initialize total to 0
    it = iter(D)    # make iterator of the deck
    while True:
        c = next(it)     # draw the next card
        total += int(c) # add card as integer to total
        print(f'your card: {c}, total = {total}.', end='')
        if total > 21:
            print(f'you lose! total = {total}')
            break
        if total == 21:
            print('you win! total = 21')
            break
        ans = input('More cards? [y/n] ')
        if ans not in {'Y', 'y'}:
            # draw one more and test
            c = next(it)
            print(f'next card is {c}. You '+ \
                  ('win' if total + c > 21 else 'lose'))
            break
```

Example session of BlackJack

```
>>> BlackJack()
your card: ♣4, total = 4. more card?  [y/n] y
your card: ♠8, total = 12. more card?  [y/n] y
your card: ♠5, total = 17. more card?  [y/n] y
your card: ♦5, total = 22. you lose! total = 22
>>> BlackJack()
your card: ♣2, total = 2. more card?  [y/n] y
your card: ♦K, total = 12. more card?  [y/n] y
your card: ♠8, total = 20. more card?  [y/n] n
next card is ♦9.  You win
>>> BlackJack()
your card: ♠4, total = 4. more card?  [y/n] y
your card: ♣3, total = 7. more card?  [y/n] y
your card: ♣A, total = 8. more card?  [y/n] y
your card: ♠6, total = 14. more card?  [y/n] y
your card: ♦Q, total = 24. you lose! total = 24
```

Generator in Python3

- Two ways to go back to caller
  - `return`
    - => activation record is destroyed
      - corresponding to StopIteration in iterator
  - `yield`
    - => activation record is kept so it can continue where it left off!
    - corresponding to next() in iterator

Example generator: fibonacci number

```
def fib():
    yield 0  # for n = 0
    yield 1  # for n = 1
    fn_minus_2 = 0
    fn_minus_1 = 1
    n = 2
    while True:
        fn = fn_minus_2 + fn_minus_1
        yield fn
        fn_minus_2, fn_minus_1 = fn_minus_1, fn
```

```
>>> f = fib()
>>> next(f), next(f), next(f), next(f), next(f), next(f)
(0, 1, 1, 2, 3, 5, 8, 13)
>>> next(f)
21
```

looks a lot like an iterator with use of next()!!!

How a generator works

- initial call to function is actually to get the generator object, not to get return value!
  - f = fib() # think "instantiate" a generator
- To run it, call next() like an iterator
  - next(f) will "continue" where the generator left off until the next yield or return statement
  - yield => save the place to continue;
    return => will not come back! (optional)

Difference between iterator and generator

- iterator
  - defined as a class
  - implements `__next__`
  - `raise` StopIteration
    ```
    try:
        next()
    except StopIteration
    ```
  - iterates an iterable
    - can be unpacked
    - directly convertible to list
  - can be used in for loop

- generator
  - defined as a function
  - uses `yield` for next
  - `return` from function
    ```
    try:
        next()
    except StopIteration
    ```
  - not tied to iterable
    - **cannot** be unpacked
    - list comprehension
  - can be used in for loop

Recursive generator

- recursive call using for-loop (easiest)
  - base case or finished => return
  - intermediate results => yield
- that is, one generator can make other generators recursively
  - must pass yielded values all the way to original caller

Example recursive generator: atom_gen()

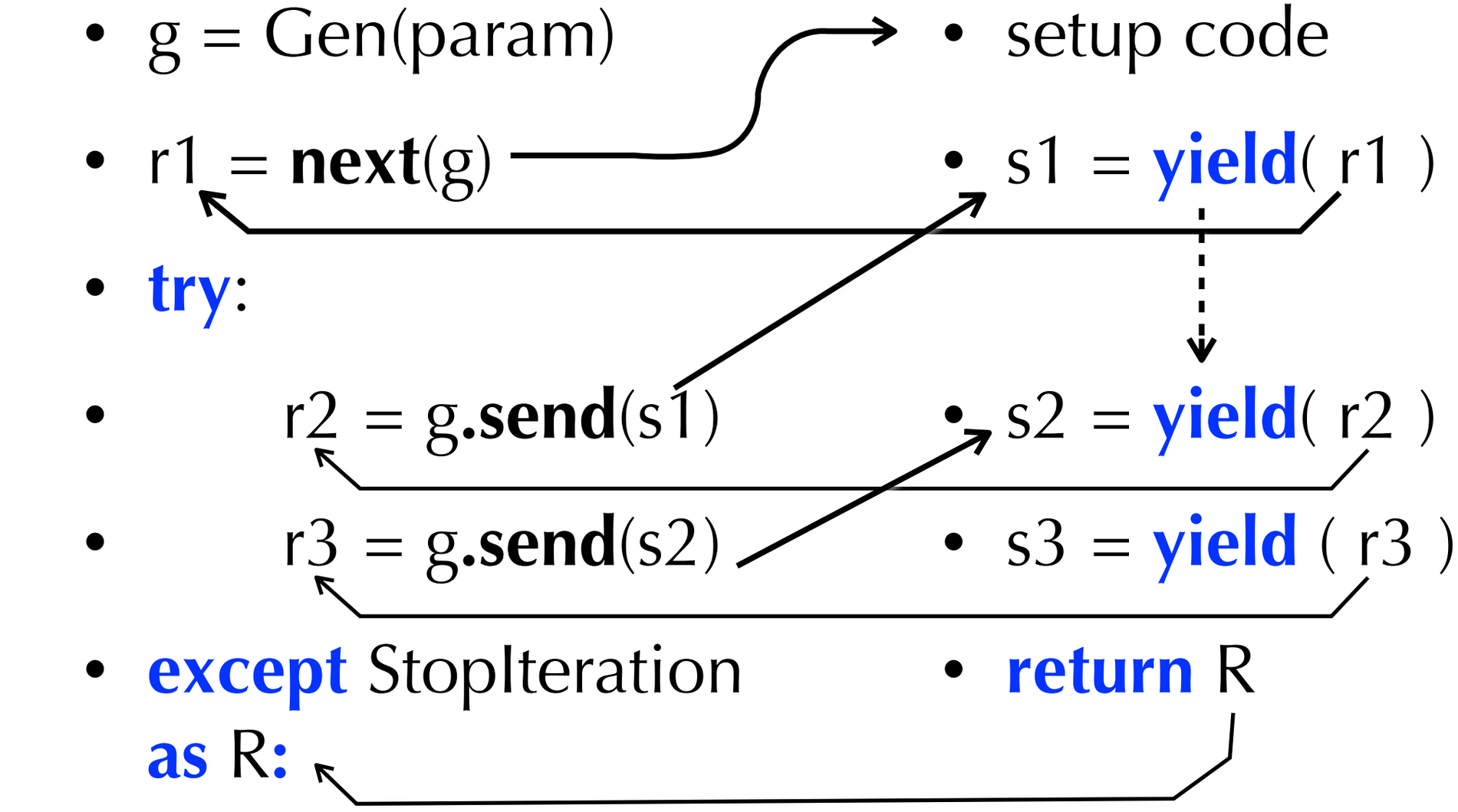
```
def atom_gen(L):
    if L is None:    # base case
        return       # raises StopIteration
    if type(L) in {list, tuple}: # recursive
        for child in L:
            # recursively generate each child of L
            for atom in atom_gen(child):
                # anything yielded must be atom
                yield atom
    else:  # base case
        yield L
```

```
>>> L = ['F1', ['F4', 'F5', ['F8']], 'F2', 'F3', 'D3', ['F6', 'F7']]
>>> [i for i in atom_gen(L)]
['F1', 'F4', 'F5', 'F8', 'F2', 'F3', 'D3', 'F6', 'F7']
```

Note: use **list comprehension** to create a list from generator output, because it is a kind of for loop, rather than list(iterable)

sending to generator and receiving in generator

- Caller
  - g = Gen(param)
  - r1 = **next**(g)
  - **try**:
  - r2 = g.**send**(s1)
  - r3 = g.**send**(s2)
  - **except** StopIteration **as** R:
- Gen (param)
  - setup code
  - s1 = **yield**(r1)
  - s2 = **yield**(r2)
  - s3 = **yield** (r3)
  - **return** R



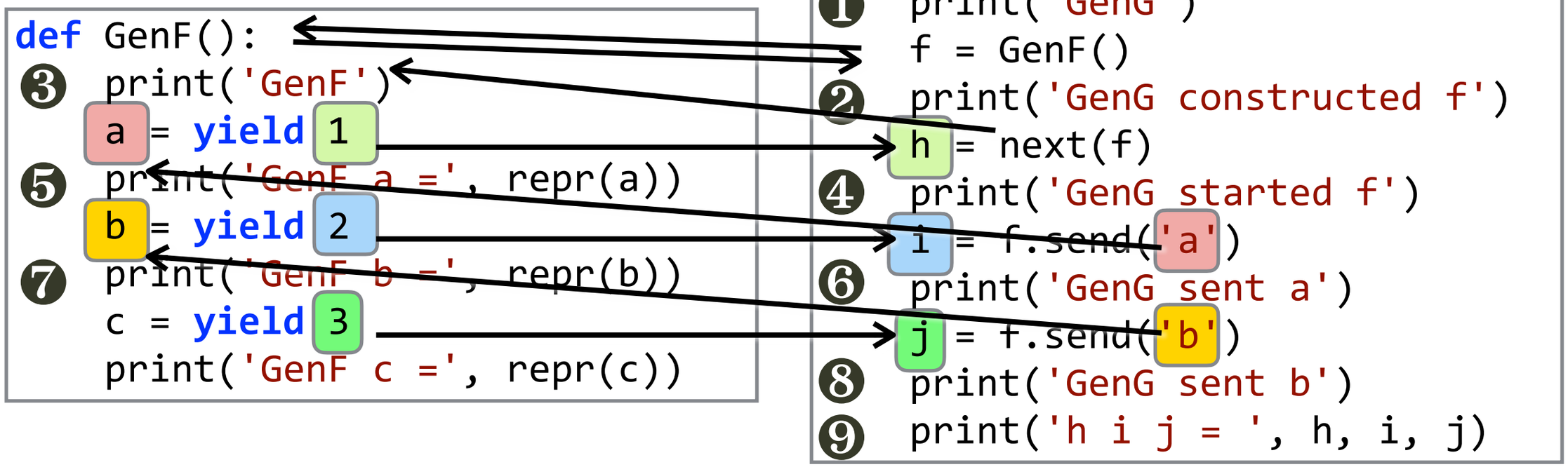
Illustrative example

```
def GenF():
❸   print('GenF')
    a = yield 1
❺   print('GenF a =', repr(a))
    b = yield 2
❼   print('GenF b =', repr(b))
    c = yield 3
    print('GenF c =', repr(c))
```

```
def GenG():
❶   print('GenG')
    f = GenF()
❷   print('GenG constructed f')
    h = next(f)
❹   print('GenG started f')
    i = f.send('a')
❻   print('GenG sent a')
    j = f.send('b')
❽   print('GenG sent b')
❾   print('h i j = ', h, i, j)
```

```
>>> GenG()
❶ GenG
❷ GenG constructed f
❸ GenF
❹ GenG started f
❺ GenF a = 'a'
❻ GenG sent a
❼ GenF b = 'b'
❽ GenG sent b
❾ h i j =  1 2 3
```

Illustrative example




```
>>> GenG()
❶ GenG
❷ GenG constructed f
❸ GenF
❹ GenG started f
❺ GenF a = 'a'
❻ GenG sent a
❼ GenF b = 'b'
❽ GenG sent b
❾ h i j =  1 2 3
```

Generator version of BlackJack: Separate Dealer and Player

- Dealer
  - instantiates Deck and shuffle
  - instantiates Player, draws & sends one card at a time
- Player
  - receives card from yield, tracks total as long as total < 21
  - yields 'yes'/'no' more card, or 'won'/'lost'
- Dealer
  - base on Player response, yes => loop, 'won'/lost' => stop, or 'no' => draw one more an decide

Generator version of Blackjack: Dealer

```
def Dealer():
    player = Player()
    D = Deck()
    D.shuffle()
    it = iter(D)
    ans = next(player) # let player run, expect hardwired "yes"
    total = 0
    while ans == 'yes':
        c = next(it)  # draw next card
        total += int(c)
        print(f'your card: {c}, ', end='')
        ans = player.send(c)
    if ans in {'lost', 'won'}:
        print(f'you {ans}')
    else:          # draw one more card and see
        c = next(it)
        print(f'next card: {c}, you '+('lost' if total+int(c) <= 21 \
                                       else 'won'))
```

Generator version of Blackjack: Player

```
def Player():
    c = yield('yes')
    total = int(c)
    while total < 21:
        ans = input(f'total = {total}, more card? [y/n] ')
        if ans not in {'Y', 'y'}: # assume no if not yes
            yield("no")  # does not come back
        else:
            c = yield("yes")
            total += int(c)
    # either we reached or exceeded 21, so no more card
    yield 'lost' if total > 21 else 'won'
```

Generator Blackjack: sample run

```
$ python -i bjgen.py
>>> Dealer()
your card: ♦A, total = 1, more card? [y/n] y
your card: ♣7, total = 8, more card? [y/n] y
your card: ♣3, total = 11, more card? [y/n] y
your card: ♦4, total = 15, more card? [y/n] y
your card: ♣10, you lost
>>> Dealer()
your card: ♠10, total = 10, more card? [y/n] y
your card: ♦5, total = 15, more card? [y/n] y
your card: ♦A, total = 16, more card? [y/n] y
your card: ♠2, total = 18, more card? [y/n] n
next card: ♦3, you lost
>>> Dealer()
your card: ♥5, total = 5, more card? [y/n] y
your card: ♠Q, total = 15, more card? [y/n] n
next card: ♣8, you won
>>> Dealer()
your card: ♠A, total = 1, more card? [y/n] y
your card: ♣J, total = 11, more card? [y/n] y
your card: ♦3, total = 14, more card? [y/n] y
your card: ♥6, total = 20, more card? [y/n] n
next card: ♠3, you won
```

Review of Iterables, Iterators, and Generators

- Ways of producing items one at a time
  - Iterable = data content (list, str, set, range(), ..)
  - Iterator = "cursor" that tracks iterable's state
  - Generator = function that "yields"
    (a way to make an object without instantiating from class!)
- Purposes
  - allowing program to be structured like for-loop, unpacking assignment / parameters (iterators)
  - performance optimization - lazy or eager evaluation (e.g., disk, network, compression)

Summary: Iterables

- Calling `iter()` on iterable to get an iterator
  - iterable can specify iterator to use by `__iter__`, or
  - iterable can let Python's iterator call `__getitem__`
- `iter()` called by for-loop, unpacking
  - user only sees the iterable; does not need to see iterator
- `r = iter(`*iterable*`)` called by user
  - call `next(`*r*`)` to get the subsequent item
  - StopIteration exception when out of items

Summary: Generator

- Caller of generator
  - g = generator() looks like a function call but makes generator
  - `next(g)` to start the generator and resume generator
  - r = g.`send(s)` to resume while sending value generator
- Generator code
  - **`yield`**`( r )` transfers control back to caller, can resume
  - s = **`yield`**`( r )` receive the value sent by subsequent g.`send(s)`
  - **`return`** (implicit or explicit) causes StopIteration