Representation

Announcements

String Representations

String Representations

In Python, all objects produce two string representations:

- The **str** is (often) legible to **humans** & shows up when you **print**
- The **repr** is (often) legible to **Python** & shows up when you **evaluate** interactively

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

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

If a type only defines a repr string, then the repr string is also the str string.

(Demo)

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
`__str__`	Method invoked by str() and print()
`__repr__`	Method invoked to display an object as a Python expression
`__eq__`	Method invoked by ==, to compare two objects
`__bool__`	Method invoked to convert an object to True or False

```
>>> t0 = Transaction(0, 20, 5)
>>> t1 = Transaction(1, 5, 5)
>>> str(t1)
'1: no change'
>>> t0 == t1
False
>>> bool(t0)
True
```

Same behavior using methods

```
>>> t1.__str__()
'1: no change'
>>> t0.__eq__(t1)
False
>>> t0.__bool__()
True
```

Class Practice

(Modified) Spring 2023 Midterm 2 Question 2(a)

```
class Letter:
    def __init__(self, contents: str):

        self.contents = contents

        self.sent = False

    def send(self):

        if self.sent:

            print(self, 'was already sent.')

        else:
            print(self, 'has been sent.’)

            self.sent = True

            return Letter(self.contents.upper())

    def __repr__(self):
        return f'Letter({repr(self.contents)})'
```

Implement the **Letter** class. A **Letter** has two instance attributes: **contents** (a **str**) and **sent** (a **bool**). Each **Letter** can only be sent once. The **send** method prints whether the letter was sent, and if it was, returns the reply, which is a new **Letter** instance with the same contents, but in all caps.
Hint: 'hi'.upper() evaluates to 'HI'.

```
"""A letter receives an all-caps reply.

>>> hi = Letter('Hello, World!')
>>> hi.send()
Letter('Hello, World!') has been sent.
Letter('HELLO, WORLD!')
>>> hi.send()
Letter('Hello, World!') was already sent.
>>> Letter('Hey').send().send()
Letter('Hey') has been sent.
Letter('HEY') has been sent.
Letter('HEY')
"""
```

(Modified) Spring 2023 Midterm 2 Question 2(b)

```
class Numbered(Letter):

    number = 0

    def __init__(self, contents):

        super().__init__(contents)

        self.number = Numbered.number
        _____________________________

        Numbered.number += 1
        _____________________

    def __repr__(self):

        return f'#{self.number}: {super().__repr__()}'
```

Implement the **Numbered** class. A **Numbered** letter has a **number** attribute equal to how many numbered letters have previously been constructed. This **number** appears in its **repr** string. Assume **Letter** is implemented correctly.

```
"""A numbered letter has a different
repr method that shows its number.

>>> hey = Numbered('Hello, World!')
>>> hey.send()
#0: Letter('Hello, World!') has been sent.
Letter('HELLO, WORLD!')
>>> Numbered('Hi!').send()
#1: Letter('Hi!') has been sent.
Letter('HI!')
>>> hey
#0: Letter('Hello, World!')
"""
```

Dictionary/Recursion Practice

Make Change

coins is a dictionary from denominations to counts. Two nickels and a quarter is {5: 2, 25: 1}

remove_one(coins, amount) returns a dictionary with one fewer count:

remove_one({5: 2, 25: 1}, 5) -> {5: 1, 25: 1}   **remove_one**({5: 2, 25: 1}, 25) -> {5: 2}

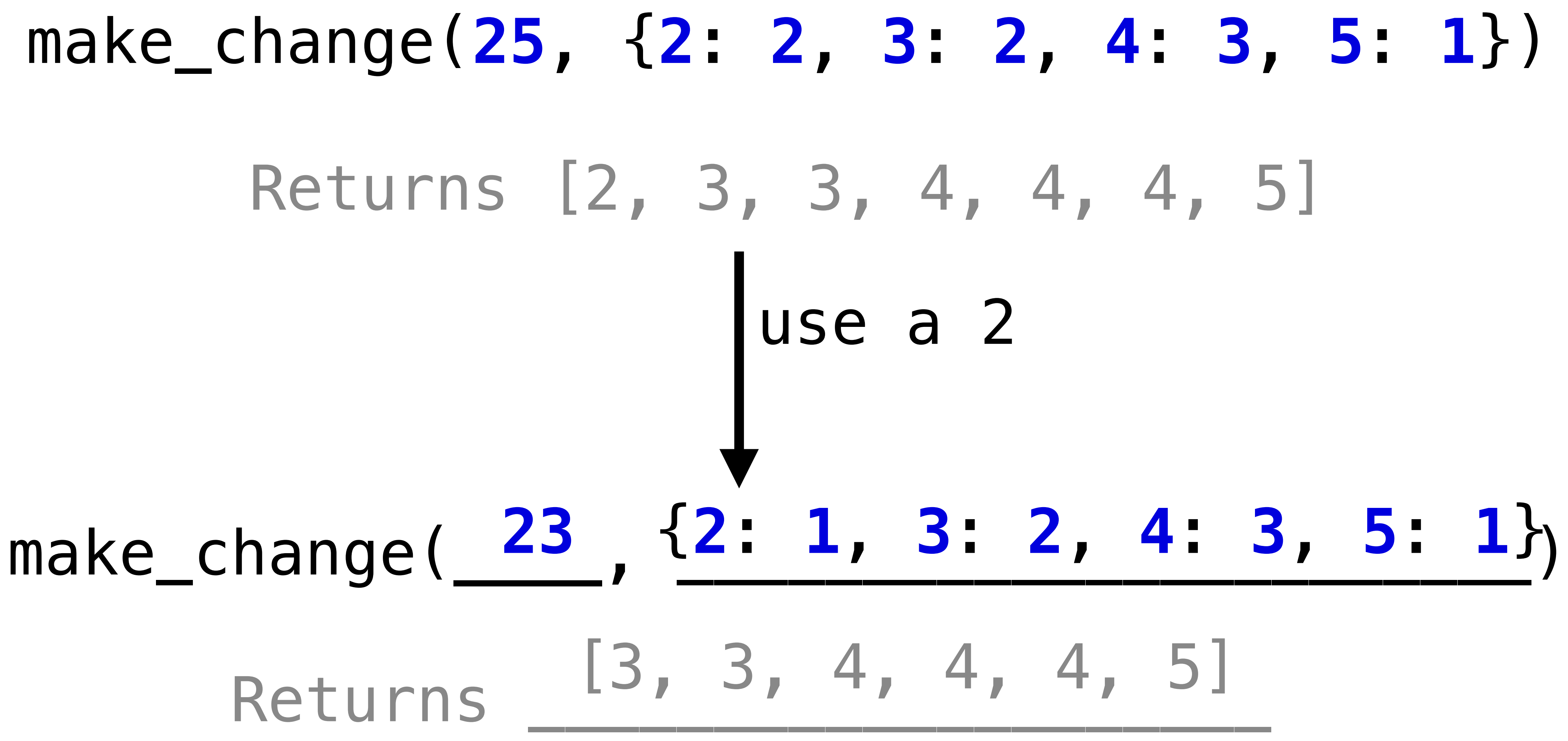
```
def make_change(amount, coins):
    """Return a list of coins that sum to amount, preferring the smallest coins
    available and placing the smallest coins first in the returned list."""
```

- What **small initial choice** can I make?

Use a 2 or don't use a 2

- What **recursive call for each option**?

```
>>> coins = {2: 2, 3: 2, 4: 3, 5: 1}
>>> make_change(8, coins)
[2, 2, 4]
>>> make_change(25, coins)
[2, 3, 3, 4, 4, 4, 5]
```



make_change(25, {2: 2, 3: 2, 4: 3, 5: 1})

Returns [2, 3, 3, 4, 4, 4, 5]

use a 2

make_change(23, {2: 1, 3: 2, 4: 3, 5: 1})

Returns [3, 3, 4, 4, 4, 5]

Make Change

coins is a dictionary from denominations to counts. Two nickels and a quarter is {5: 2, 25: 1}

remove_one(coins, amount) returns a dictionary with one fewer count:

remove_one({5: 2, 25: 1}, 5) -> {5: 1, 25: 1} **remove_one**({5: 2, 25: 1}, 25) -> {5: 2}

```
                   25    {2: 2, 3: 2, 4: 3, 5: 1}
def make_change(amount, coins):
    """Return a list of coins that sum to amount, preferring the smallest coins
    available and placing the smallest coins first in the returned list."""
    if not coins:
        return None
    smallest = min(coins) smallest is 2
    rest = remove_one(coins, smallest)
    if amount < smallest: rest is {2: 1, 3: 2, 4: 3, 5: 1}
        return None
    elif amount == smallest:

        return [smallest]
    else:                               23
        result = make_change(amount-smallest, rest)   result is [3, 3, 4, 4, 4, 5]
        if result:
            return [smallest] + result  [2] + [3, 3, 4, 4, 4, 5] -> [2, 3, 3, 4, 4, 4, 5]
        else:
            return make_change(amount, rest)
```

```
>>> coins = {2: 2, 3: 2, 4: 3, 5: 1}
>>> make_change(8, coins)
[2, 2, 4]
>>> make_change(25, coins)
[2, 3, 3, 4, 4, 4, 5]
```

```
make_change(23, {2: 1, 3: 2, 4: 3, 5: 1})
    Returns [3, 3, 4, 4, 4, 5]
```