

3 Polynomials

We will start this problem by implementing two functions to perform operations on polynomials: `poly_add(p1, p2)` and `poly_mul(p1, p2)`. For both functions, the inputs `p1` and `p2` each represent polynomials, but we are not told the details of the actual representation of `p1` and `p2`; rather, we are given descriptions of various helper functions that operate on polynomials (this list is also reproduced on page 19 of the exam, which you may remove):

- `zero_poly()` returns a new polynomial representing 0.
- `get_order(poly)` returns the order of the polynomial `poly`. For example, if the input represented $8 + 7x + 4x^3$, this function would return 3. Assume that the order of the zero polynomial is -1 .
- `get_coeff(poly, i)` returns the coefficient associated with the x^i term in the given polynomial. For example, if we called `get_coeff(p, 3)` where `p` represented $8 + 7x + 4x^3$, this function would return 4; and calling `get_coeff(p, 2)` would return 0.
- `set_coeff(poly, i, val)` *mutates* the given polynomial such that the coefficient associated with x^i is replaced with `val`. For example, calling `set_coeff(p, 2, 9)` where `p` represented $8 + 7x + 4x^3$ would result in the input polynomial being mutated to represent $8 + 7x + 9x^2 + 4x^3$.
- `shifted(poly, n)` returns a *new* polynomial representing the given `poly` multiplied by x^n . For example, calling `shifted(p, 4)` where `p` represented $8 + 7x + 4x^3$ would result in a new polynomial $8x^4 + 7x^5 + x^7$.
- `scaled(poly, n)` returns a *new* polynomial representing the given `poly` with all coefficients scaled by `n`. For example, calling `scaled(p, 4)` where `p` represented $8 + 7x + 4x^3$ would result in a new polynomial $32 + 28x + 16x^3$.

To start, implement `poly_add` below. This function should take two polynomials as input, and it should return a new polynomial representing the sum of the two input polynomials, without mutating either of the inputs. For example, if `p1` represents $3x + 7$ and `p2` represents $2x^2 + 9x$, then `poly_add(p1, p2)` should return a new polynomial representing $2x^2 + 12x + 7$. You should not make any assumptions about the exact representations of `p1` or `p2`, but you may assume working implementations of all of the helper functions described above.

```
def poly_add(p1, p2):
    out = zero_poly()
    for power in range(max(get_order(p1), get_order(p2))+1):
        set_coeff(out, power, get_coeff(p1, power) + get_coeff(p2, power))
    return out
```

Next, we'll implement `poly_mul`. This function should take two polynomials as input, and it should return a new polynomial representing the *product* of the two input polynomials, without mutating either of the inputs. For example, if `p1` represents $3x + 7$ and `p2` represents $2x^2 + 9x$, then `poly_mul(p1, p2)` should return a new polynomial representing $6x^3 + 41x^2 + 63x$. Write your code for `poly_mul` in the box below. You may assume working implementations of all of the helper functions from the previous page, as well as a working implementation of `poly_add`, but you should not define any additional helper functions here. As before, your code should not make assumptions about the exact representations of `p1` or `p2`.

```
def poly_mul(p1, p2):  
    out = zero_poly()  
    for ix1 in range(get_order(p1)+1):  
        new = shifted(scaled(p2, get_coeff(p1, ix1)), ix1)  
        out = poly_add(out, new)  
    return out
```

As we have seen throughout 6.101, the choice of representation is an important one! Careful choice of internal representation can often allow us to write more concise, and clearer code; and this problem is no exception! While your code for `poly_add` and `poly_mul` should work for *any* internal representation, we'll now try to fill in the details.

Here the choice of internal representation is up to you. You are welcome to use any combination of Python `int`, `float`, `str`, `bool`, `list`, `tuple`, `set`, `frozenset`, and/or `dict` objects; but you are not welcome to use classes (which have not yet been covered in 6.101). Your grade for this problem will depend on the correctness of your implementation. Your code will not be graded for efficiency in terms of time or memory, but note that some representations may be harder to implement correctly.

In the box below, briefly describe your choice of representation:

There are many possibilities here, but these solutions will use a dictionary mapping powers to their associated coefficients. For example, $5x^4 + 6x^3 + 27$ will be represented as `{4: 5, 3: 6, 0: 27}`.

Then, in the box below and the box on the facing page, implement the `zero_poly`, `get_order`, `get_coeff`, `set_coeff`, `shifted`, and `scaled` helper functions using that representation.

```
# your polynomial helper functions here
def zero_poly():
    return {}

def get_order(poly):
    return max((k for k in poly if poly[k] != 0), default=-1) # make sure to ignore 0s!

def get_coeff(poly, i):
    return poly.get(i, 0)

def set_coeff(poly, i, val):
    poly[i] = val

def shifted(poly, n):
    return {k+n: v for k, v in poly.items()}

def scaled(poly, n):
    return {k: v*n for k, v in poly.items()}
```

```
# your polynomial helper functions here
# or, a different representation involving lists
# here, p[i] is the coefficient associated with the x**i term
# we'll assume that trailing zeros NEVER exist in the representation
# (and we'll set up the code to make sure that's always the case)

def zero_poly():
    return []

def get_order(poly):
    # again, we need to be careful not to count explicit zeros toward
    # our order calculation; but in this case, we handle that in set_coeff
    # instead
    return len(poly) - 1

def get_coeff(poly, i):
    return poly[i] if 0 <= i < len(poly) else 0

def set_coeff(poly, i, val):
    while get_order(poly) < i:
        poly.append(0)
    poly[i] = val
    while poly and poly[-1] == 0:
        poly.pop()

def shifted(poly, n):
    return [0]*n + poly

def scaled(poly, n):
    return [coeff * n for coeff in poly]
```

3 GPT

You've been asked to write a function to find the first sentence in a Python string, where a sentence is defined to be a sequence of adjacent characters that starts with a capital letter, ends with a period, and doesn't contain any other periods. If there are no sentences in the given input, your function should return `None` instead.

But it's a nice day outside and you'd rather not spend it sitting in front of your computer, so you ask ChatGPT to write it for you. ChatGPT cheerfully replies: "Certainly, here is a Python function that finds the first sentence in a Python string," and then it produces the following code:

```
def find_sentence(string):
    start = 0
    i = start
    scanning = False
    while i < len(string)-1:
        if string[i].isupper() and not scanning:
            start = i
            scanning = True
        elif string[i] == '.' and scanning:
            return string[start:start+i]
        i += 1
```

To test it out, you run the code through a small test case:

```
>>> find_sentence('not a sentence. This is a sentence. also not a sentence.')
'This is a sentence. also not a sen'
```

Oh, no! Despite ChatGPT's cheerful response, the code is broken. The test case above should have returned 'This is a sentence.' instead. Seeing this behavior, you decide to test the code further.

On the facing page are several examples of results that the code could have when it is run. For each, if it is possible for the code above to produce that result, provide an input that would lead to that result. If an outcome is not possible, write an "X" in the box instead. Your inputs should all be distinct from the example given above, but they should all be valid (i.e., they should be strings).

Valid input that produces the correct string as a result (or X if not possible):

"aSentence. wheee"

(there must be a single character before the first sentence, and the period can't be the least character)

Valid input that correctly produces None as a result (or X if not possible):

'no sentences here.'

(any string with no sentence in it)

Valid input that should produce a string but produces an incorrect string (or X if not possible):

'this is like. The example. from above.'

(any string where the first sentence starts at an index other than 1)

Valid input that should produce a string but instead produces None (or X if not possible):

'hey, This is my sentence.'

(any string where nothing follows the first sentence)

Valid input that should produce None but instead produces a string (or X if not possible):

X

(this can't happen; we only return a string if there is a sentence)

Valid input that causes an exception to be raised (or X if not possible):

X

(this can't happen for well-formed input)

Valid input that enters an infinite loop (or X if not possible):

X

(this can't happen either)

Problem \times 11

Consider the code below:

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

def apply(f, m):
    for i in range(m):
        n = m
        f(i)
```

`apply(fact, 3)`

After this code runs to completion, how many frames have been created that contain a binding for the name `n`?

7

Briefly show your work by saying which frames you counted. Your answer should fit in the box without scrolling.

`apply()` is called once, and its frame will contain `n`. It then calls `fact(0)`, `fact(1)`, and `fact(2)`, and the recursive computation of `fact(n)` creates `n+1` frames each containing the variable `n`, for a total of $1 + 2 + 3 = 6$ frames. Adding the frame for `apply` gives 7.

After running the code below:

```
a = [1,2,3]
b = ['A'] + a + ['z']
c = { 'w': a }
d = a[0:3]
e = [a,a,a]
e.reverse()
f = [a[0], a[1], a[2]]
g = [[4],[a],[6]][1]
```

Write three different expressions that return an alias of `a`. None of the expressions should use the variable `a` itself, and each expression must use different variables: if one expression uses `b`, for example, the other two expressions may not.

`c['w']`

`e[0]` (or 1 or 2)

`g[0]`

Which of the variables above *cannot* be used to get an alias of `a`? Give a brief reason why, for each one. Your answer should fit in the box without scrolling.

`b` cannot, because concatenation of lists copies the elements.

`d` cannot, because a slice copies the elements.

`f` cannot, because it makes a fresh list containing the three elements of `a`.

After running the code below:

```
log = []
def monitor():
    x = 4
    def log_x():
        log.append(x)
    x = 5
    log_x
    x = 6
    return log_x
log_x = monitor()
x = 7
log_x()
x = 8
(lambda : log_x)()
```

What is the value of `log` after the code runs to completion?

`[6]`

(This is the end of the exam.)

2.2 Program 2

```
x = 307

class A:
    x = 308

B = A

class C(B):
    x = 309
    def __init__(self):
        self.x = x

class D(C):
    def __init__(self):
        pass

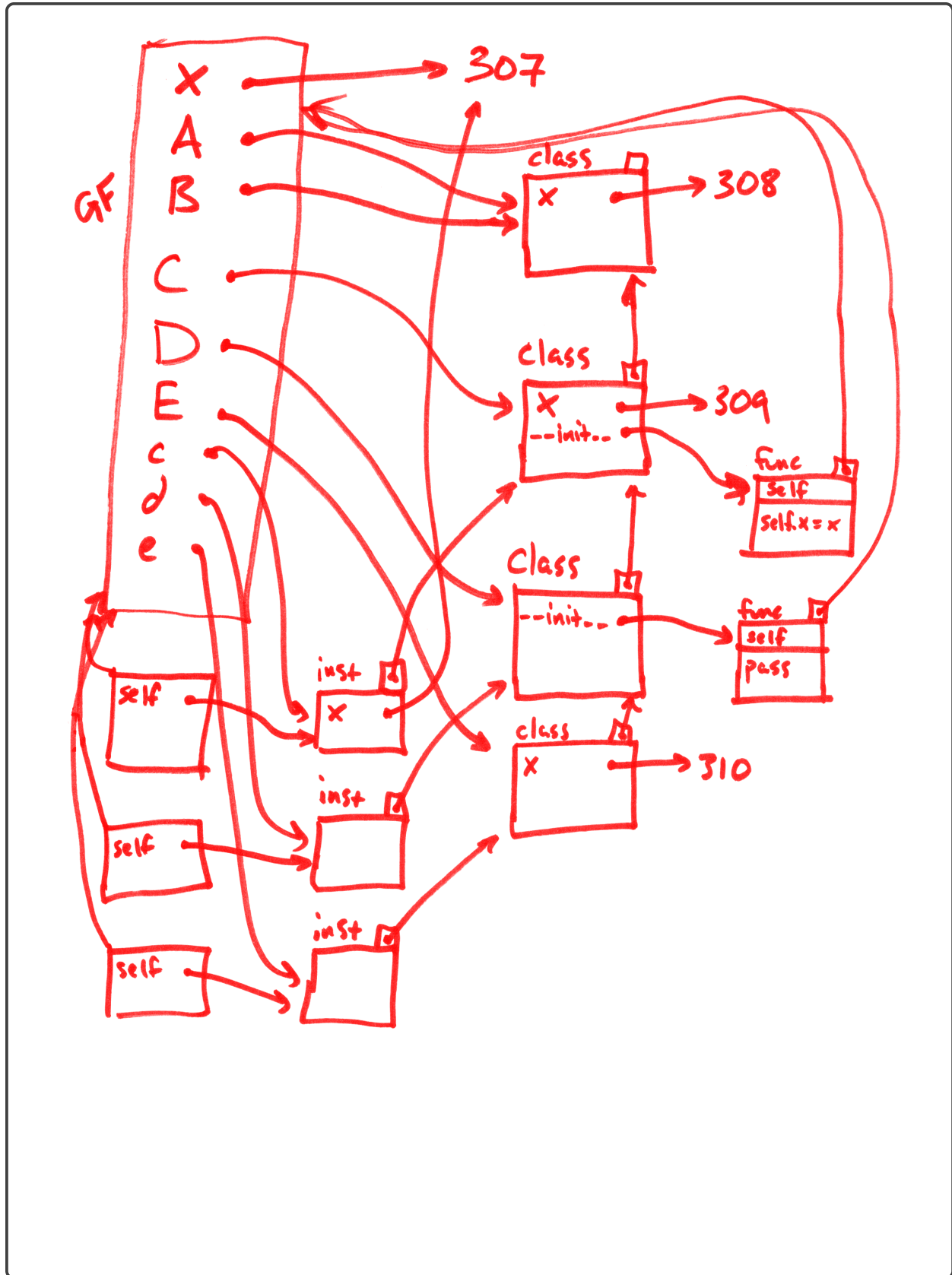
class E(D):
    x = 310

c = C()
d = D()
e = E()
print(c.x)
print(d.x)
print(e.x)
```

Program 2 Output:

```
307
309
310
```

Program 2 Environment Diagram:



3 Scoping out your Inheritance

Write what will get printed for each of these code sequences. If running the code sequence would result in an error, write ERROR instead.

Part a:

```
class A:
    foo = 1

    def update(self, i):
        self.foo = self.foo + i

a = A()
a.update(10)
print(a.foo, A.foo)
```

11 1

Part b:

```
class A:
    foo = 1

    def update(self, i):
        self.foo += i

a = A()
a.update(10)
print(a.foo, A.foo)
```

11 1

Part c:

```
class A:
    foo = [1]

    def update(self, i):
        self.foo = self.foo + [i]

a = A()
a.update(10)
print(a.foo, A.foo)
```

[1, 10] [1]

Part d:

```
class A:
    foo = [1]

    def update(self, i):
        self.foo += [i]

a = A()
a.update(10)
print(a.foo, A.foo)
```

[1, 10] [1, 10]

Part e:

```
class A:
    foo = [1]

    def update(self, i):
        self.foo.extend([i])

class B(A):
    foo = [100]

    def update(self, i):
        self.foo.extend([i, i])

b = B()
b.update(10)
print(b.foo, A.foo, B.foo)
```

```
[100, 10, 10] [1] [100, 10, 10]
```

Part f:

```
class A:
    foo = [1]

    def update(self, i):
        self.foo.extend([i])

class B(A):
    foo = [100]

    def update(self, i):
        A.update(self, i)

b = B()
b.update(10)
print(b.foo, A.foo, B.foo)
```

```
[100, 10] [1] [100, 10]
```

Part g:

```
class A:
    foo = [1]

    def update(self, i):
        self.foo.extend([i])

class B(A):
    foo = [100]

    def update(self, i):
        A.update(self, i)
        self.foo.extend([i, i])

b = B()
b.update(10)
print(b.foo, A.foo, B.foo)
```

```
[100, 10, 10, 10] [1]
[100, 10, 10, 10]
```

Part h:

```
class A:
    foo = [1]

    def update(self, i):
        A.foo.extend([i])

class B(A):
    foo = [100]

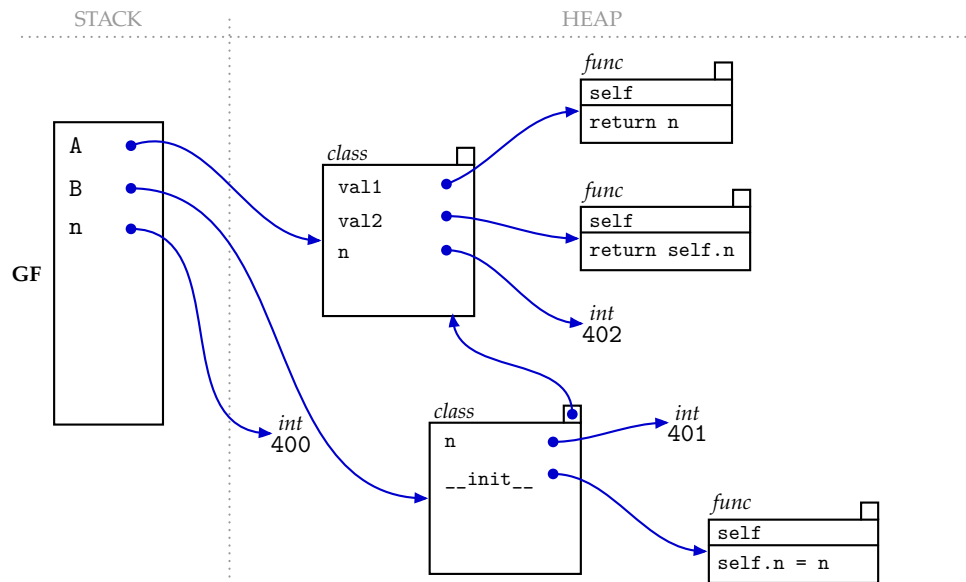
    def update(self, i):
        A.update(self, i)

b = B()
b.update(10)
print(b.foo, A.foo, B.foo)
```

```
[100] [1, 10] [100]
```

3 Environment Diagrams

The following diagram represents the state of a program after some code has been executed, purely in the global frame (no additional frames have been created as of this point in the program's execution). This diagram is almost complete, except that the enclosing frames of the functions are intentionally not shown.



Starting from this state, the following code is then run in the global frame:

```
n = 403
```

```
class C(A):
    def __init__(self):
        self.n = self.n
```

After this code is run, consider running the code on the facing page in the order it is specified. Each print statement in the code is followed by a box. For each, consider the value that would be printed by that print statement, and:

- If nothing would be printed because of an infinite loop (or infinite recursion), write *infinite* in the box.
- If nothing would be printed because of an exception other than infinite recursion, write *exception* in the box.
- Otherwise, write the printed value in the box.

```
a = A()
print(a.val1())
```

403

```
print(a.val2())
```

402

```
b = B()
print(b.val1())
```

403

```
print(b.val2())
```

403

```
c = C()
print(c.val1())
```

403

```
print(c.val2())
```

402

```
def foo():
    n = 404
    class D(B):
        pass
    n = 405
    return D()
```

```
d = foo()
print(d.val1())
```

403

```
print(d.val2())
```

403

```
def bar():
    n = 406
    class E(C):
        def __init__(self):
            self.n = n
    n = 407
    return E()
```

```
e = bar()
print(e.val1())
```

403

```
print(e.val2())
```

407