9.0 - Week 9 - Workshop (MA)

Learning Objectives

- Understanding Iterators (and *generators*)
- Understanding Linked Stacks

Week 9 Padlet Discussion Board link: https://monashmalaysia.padlet.org/fermi/2022week9

Why Iterators?

Question *Submitted Sep 19th 2022 at 11:15:58 am*

What is the purpose of iterators in programming languages?

[✓ They allow a user of a container ADT to *iterate through the items* of the container

- in a *unified* way and
- *without knowing* how the container is implemented.]

Just a weird concept that *only some* programming languages have!

Have no idea. More MIPS! I am a MIPS god!

Reiterating Iterators

Normally, iterators are implemented as *external objects* that access the structure of the container we want to iterate through. In Python, iterator objects must have two *magic methods*:

- `__iter__()`
- `__next__()`

Furthermore, method `__iter__()` must return the iterator object itself.

Besides that, the container we want to iterate through must have method `__iter__()` too *(it links the container with its iterator)*:

```
class Container():
    def __iter__(self) -> ContainerIterator:
        return ContainerIterator(self.something)
```

```
class ContainerIterator():
    def __iter__(self) -> ContainerIterator:
        return self

    def __next__(self) -> SomeElementType:
        if THERE_IS_NEXT_ELEMENT:
            return NEXT_ELEMENT  # of type SomeElementType
        else:
            raise StopIteration
```

Let's implement an iterator for our LinkedList!

Iterators and Sequences

So far, we've always seen **Iterators** when they're paired with **containers**. But this doesn't have to be the case. We can define iterators that produce sequences of values directly:

```
class SequenceIterator:
    def __init__(self):
        self.current = 0
    def __iter__(self):
        return self
    def __next__(self):
        elt = self.current
        self.current += 2
        return elt

iter = SequenceIterator()
print(next(iter))
print(next(iter))
print(next(iter))

input("What happens in a for loop?")
for value in SequenceIterator():
    print(value)
```

Now try it yourself: complete the methods in `SquareSeq` to make an iterator, which will return the first k square numbers (e.g. $1, 4, 9$, etc.)

Generators

In practice, it may be easier to use a limited form of iterators called *generators*.

Generators are objects that *look like functions* but *behave like iterators*.

The key concept of generators is the **yield statement**. It tells Python that the function is a generator. The function may ***yield*** values multiple times, each time *saving its state*.

```
class LinkedListGen(LinkedList[T]):
    """ Same LinkedList but with a simple generator-based iterator. """

    def __init__(self, dummy_capacity=1) -> None:
        """ Linked-list object initialiser. """
        LinkedList.__init__(self)

    def __iter__(self) -> T:
        """ Generator-based iterator. """
        current = self.head
        while current is not None:
            yield current.item  # yielding behaves like "__next__()"
            current = current.link
        # StopIteration is raised automatically upon reaching the end
```

Generator functions in Python raise **`StopIteration`** *automatically* as soon as no more values can be yielded. It can also be used with the **`next()`** call just like a full-blown iterator.

Example of a generator:

```
def square_seq(limit):
    for i in range(1, limit):
        yield i*i

for number in square_seq(5):
    print(number)

input("How is square_seq represented internally?")
print(dir(square_seq(5)))
```

Playing with Generators

```
def square_seq(limit):
    for i in range(1, limit):
        yield i*i

for number in square_seq(5):
    print(number)

input("How is square_seq represented internally?")
print(dir(square_seq(5)))
```

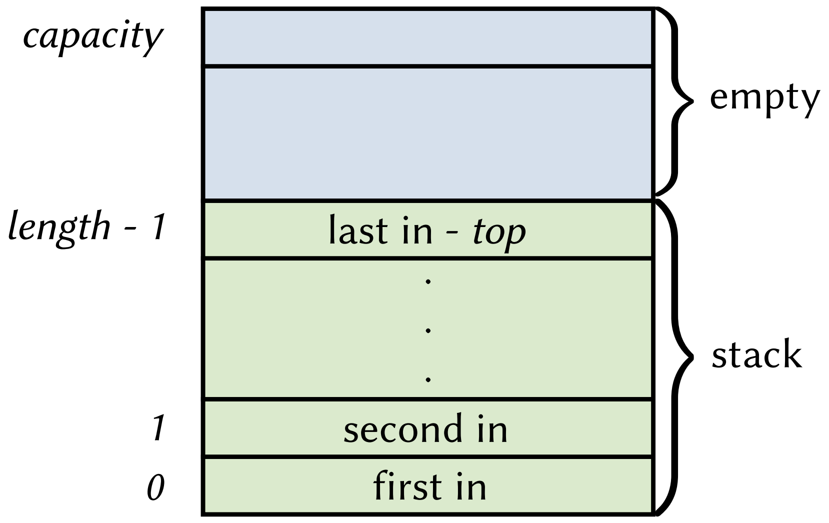
Now it's your turn: write code to produce that produce first k ***Fibonacci*** numbers:

$$1, 1, 2, 3, 5, 8, \ldots$$

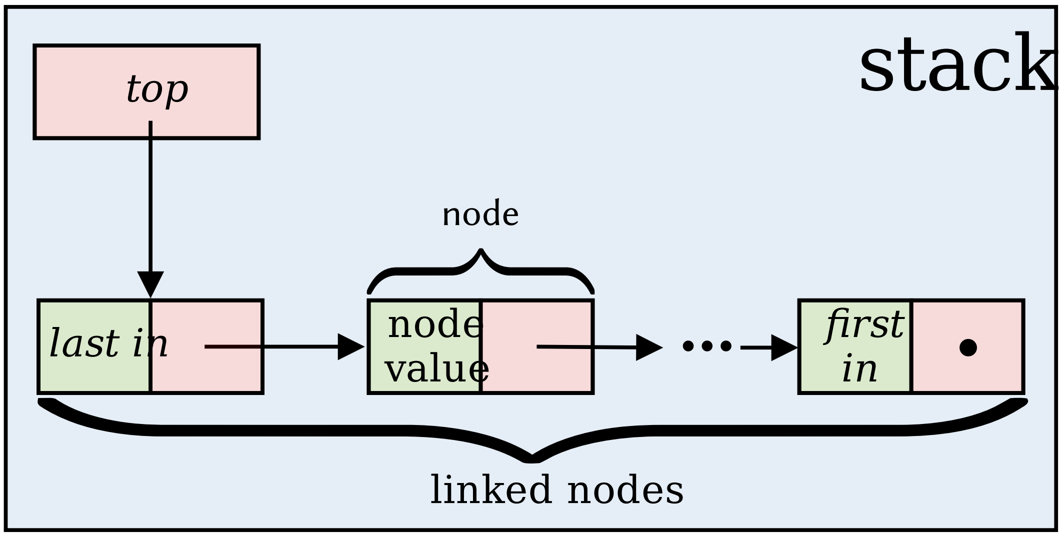
using generators.

Linked Stack vs. its Array-based Counterpart

Similar to lists (see our previous workshop), stacks can be *compactly* represented with arrays:



An alternative representation that uses *more memory* per element but *does not need to reallocate* new chunks of memory for larger arrays and *copy* the stack elements:



Example `push` operation:

```
def push(self, item: T) -> None:
    # creating a new node
    new_node = Node(item)
    # linking it
    new_node.link = self.top
    self.top = new_node
    # increasing the length
    self.length += 1
```

Example `pop` operation:

```
def pop(self) -> T:
    if self.is_empty():
        raise Exception('Stack is empty, nothing to pop!')
    # getting the item to return
    item = self.top.item
    # moving the top
    self.top = self.top.link
    # decreasing the length
    self.length -= 1
    return item
```

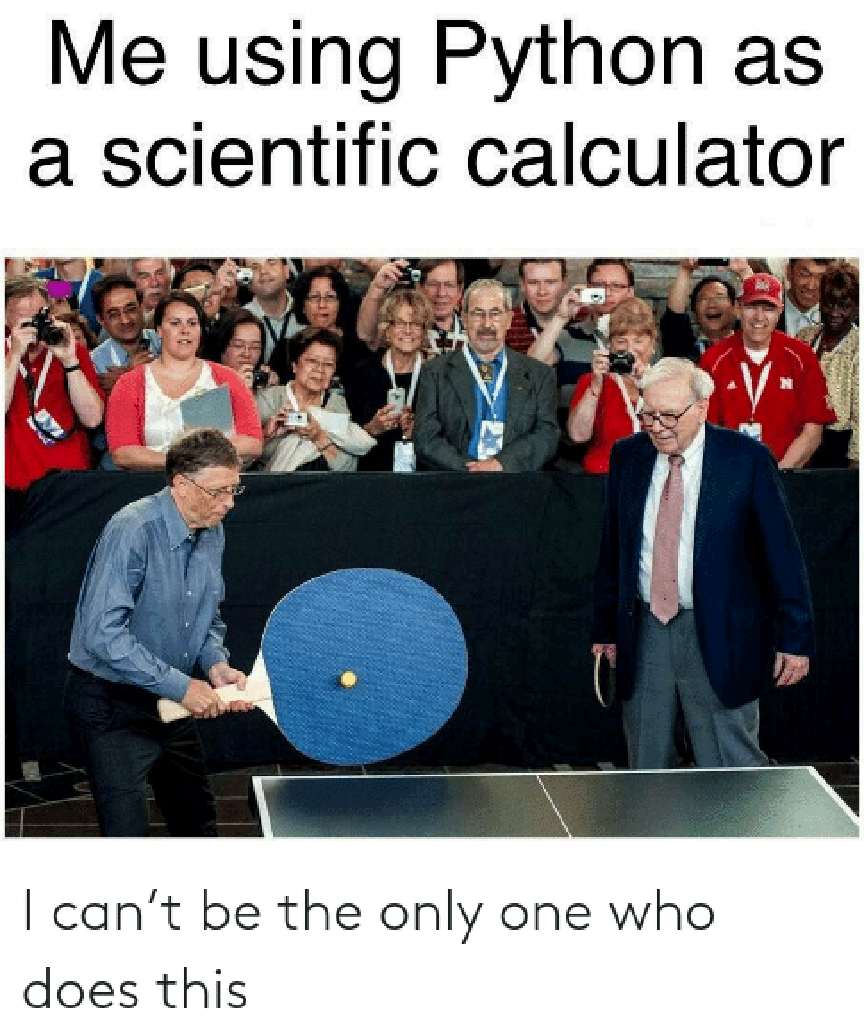
Applications of Stacks

There are several important use cases for the Stack ADT, for instance:

- memory management
- backtracking in *search algorithms*
- expression parsing and evaluation

In this exercise, we will use linked stacks to implement a simple calculator.

Given an arithmetic expression, it should *calculate the value* of the expression, e.g. `(3 + 2) * 4` should be evaluated to `20` while `3 + 2 * 4` should be evaluated to `11`.


Me using Python as a scientific calculator
I can't be the only one who does this

Simple Calculator

A typical calculator needs to:

1. parse the input expression in the **standard infix notation, e.g.** `(3 + 2) * 4`
2. convert it to the **postfix notation**, **e.g.** `3 2 + 4 *`
3. evaluate the postfix notation

All three steps can be done using stacks!

Let's focus on **step 3 only** (steps 1 and 2 are already done)

Our task is to implement a method `eval_postfix()`. How it works:

1. it receives a list representing an expression in the postfix form
2. creates a ***stack for the calculations***
3. ***traverses*** all the tokens in the postfix notation and
   - if the token is an *operand* (number), pushes it to the stack
   - if the token is an *operator*, takes two top elements and applies the operator to these elements
   - it then *pushes the result* to the stack
4. *in the end*, the ***top of the stack*** is returned as the result
   (invariant: at this point, the stack has 1 item!)

```
Example:
    infix   = A + (B - C)
    postfix = A B C - +

    1. A -> stack.push(A)
    2. B -> stack.push(B)
    2. C -> stack.push(C)

    3. - -> op1 = stack.pop()  # this returns C
            op2 = stack.pop()  # this returns B
            res = op2 - op1    # res = B - C
            stack.push(res)
    4. + -> op1 = stack.pop()  # this returns res
            op2 = stack.pop()  # this returns A
            res = op2 + op1    # res = A + res
            stack.push(res)

    *. return stack.pop()      # the stack must contain 1 element!
```

Feedback Form

Weekly Workshop Feedback Form

Question 1

I am enrolled in:

○ □□ Australia

○ □□ Malaysia

Question 2

What needs improvement?

No response

Question 3

What worked best?

No response

Question 4

How engaged were you by the workshop?

○ □□□ Very engaged

○ □□□ Engaged

○ □😐□ Not impressed

○ 😕😴□ Lost