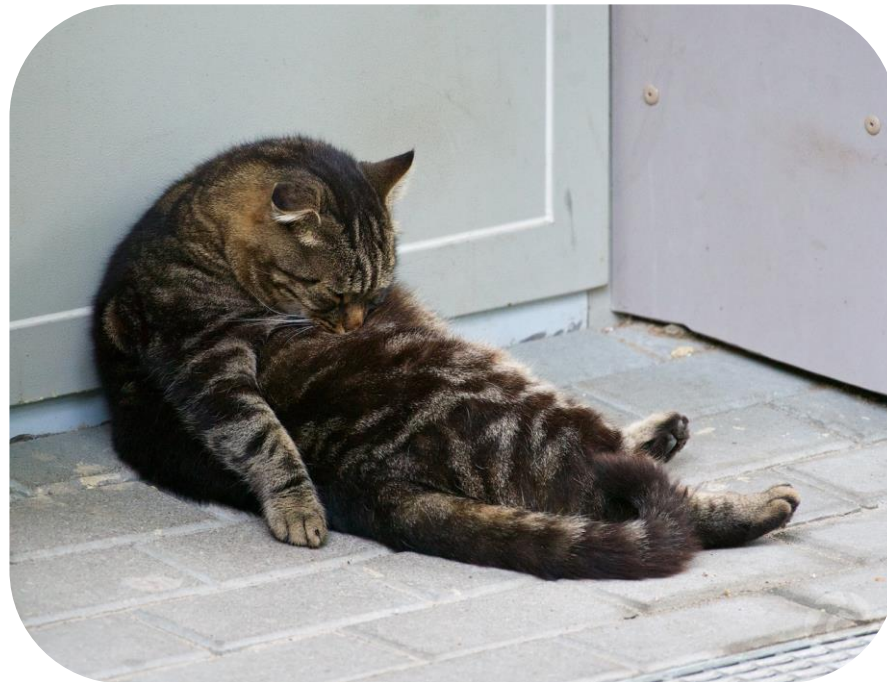


# Lecture 13

## Iterators & Generators





# Iterators



# Definitions

- Lazy evaluation - Delays evaluation of an expression until its value is needed
- Iterable - An object capable of returning its members one at a time. Examples include all sequences (lists, strings, tuples) and some non-sequence types (dictionaries).
- Iterator - An object that provides sequential access to values, one by one.
  - All iterators are iterables. Not all iterables are iterators.
- Metaphor: Iterables are books & Iterators are bookmarks

# Iterators

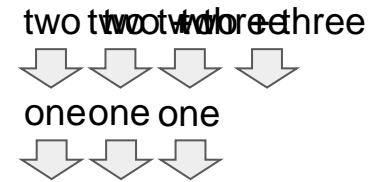
## How do we create iterators?

`iter(iterable)`: Return an iterator over the elements of an iterable value.

- This method creates a bookmark from a book starting at the front.

`next(iterator)`: Return the next element in an iterator.

- Returns the current page and moves the bookmark to the next page.
- The iterator remembers where you left off. If the current page is the end of the book, error.



```
>>> s = [1, 2, 3] # the book
>>> one, two = iter(s), iter(s)
>>> next(one) # move bookmark 1
1
>>> next(two) # move bookmark 2
1
>>> next(one) # move bookmark 1
2
>>> next(two) # move bookmark 2
2
>>> three = iter(two)
>>> next(three) # move bookmark 2 & 3
3
>>> next(two) # Ran out of pages
Stop Iteration
```

# Check Your Understanding: Fibonacci

Define a function that returns an iterator that outputs up to the nth value in the Fibonacci sequence. You can assume n will always be 2 or greater

- Remember, `iter(iterable)` creates an iterator. Lists are iterables.

```
def fib_iter(n):  
    """  
    >>> x = fib_iter(4)  
    >>> next(x)  
    0  
    >>> next(x)  
    1  
    >>> next(x)  
    1  
    >>> next(x)  
    2  
    """
```

Have you missed me?



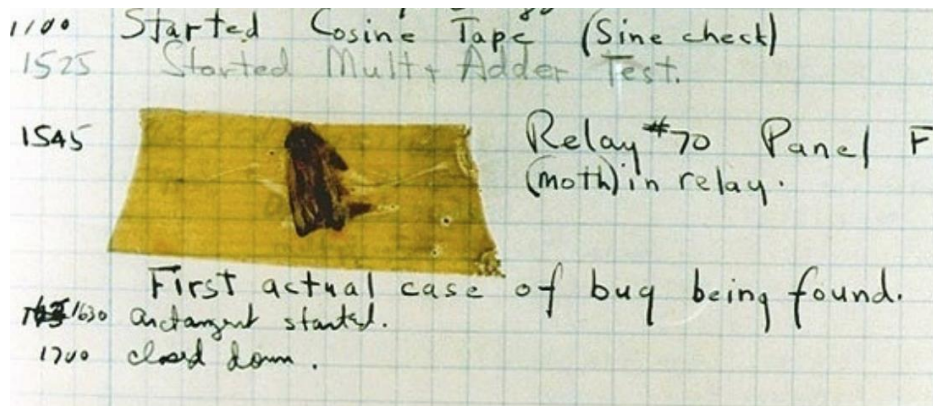


# Exceptions / Errors

# Exceptions / Errors

Sometimes, computer programs behave in non-standard ways

- A function receives a argument value of an improper type
- Some resources (such as a file) is not available
- A network connection is lost in the middle of data transmission



Grace Hopper's Notebook, 1947, Moth found in a Mark II Computer



# Raise Exceptions

Exceptions are raised with a raise statement

```
raise <expression>
```

<expression> must be an Exception, which is created like so:

E.g., `TypeError( 'Error message' )`

`TypeError` -- A function was passed the wrong number/type of argument

`NameError` -- A name wasn't found

`KeyError` -- A key wasn't found in a dictionary

`RuntimeError` -- Catch-all for troubles during interpretation

# Try Statements

Try statements handle exceptions

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
```

Execution rule:

1. The <try suite> is executed first
2. If, during the course of executing the <try suite>, an exception is raised that is not handled otherwise then the <except suite> is executed with <name> bound to the exception

```
>>>try:
... x = 1/0
... except ZeroDivisionError as e:
...     print('Except a', type(e))
...     x = 0
Except a <class 'ZeroDivisionError'>
>>> x
0
```

# Back to Iterators - The For Statement

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

1. Evaluate the header <expression>, which must evaluate to an iterable object.
2. For each element in that sequence, in order:
  - a. Bind <name> to that element in the first frame of the current environment
  - b. Execute the <suite>

When executing a for statement, `iter` returns an iterator and `next` provides each item:

```
>>> counts = [1, 2, 3]  
>>> for item in counts:  
...     print(item)  
1  
2  
3
```

These are  
equivalent!

```
>>> counts = [1, 2, 3]  
>>> items = iter(counts)  
>>> try:  
...     while True:  
...         item = next(items)  
...         print(item)  
... except StopIteration:  
...     pass  
1  
2  
3
```

StopIteration is raised whenever next is called on an empty iterator

Generators



# Definitions and Rules

Some definitions:

- Generator: An iterator created automatically by calling a generator function.
- Generator function: A function that contains the keyword `yield` anywhere in the body

When a generator function is called, it returns a generator **instead of** going into the body of the function. The only way to go into the body of a generator function is by calling `next` on the returned generator.

Yielding values are the same as returning values except `yield` remembers where it left off.

# Generators and Generator Functions

```
>>> def plus_minus(x):  
...     yield x  
...     yield -x  
  
>>> t = plus_minus(3)  
>>> next(t)  
3  
>>> next(t)  
-3  
>>> t  
<generator object>  
>>> next(t)  
StopIteration
```

We are allowed to call `next` on generators because generators are a type of iterator.

Calling `next` on a generator goes into the function and evaluates to the first `yield` statement. The next time we call `next` on that generator, it resumes where it left off (just like calling `next` on any iterator!)

Once the generator hits a return statement, it raises a `StopIteration`

# Generators to Represent Infinite Sequences

Iterators are used to represent infinite sequences. In this course, when we ask you to write an iterator, we want you to write a generator.

```
>>> def naturals():
...     x = 0
...     while True:
...         yield x
...         x += 1

>>> nats = naturals()
>>> next(nats)
0
>>> next(nats)
1
>>> nats1, nats2 = naturals(), naturals()
>>> [next(nats1) * next(nats2) for _ in range(5)]
[0, 1, 4, 9, 16] # Squares the first 5 natural numbers
```

## Check Your Understanding: Generators

Given the following generator function, what will the call to `gen()` return?

```
>>> def gen():
...     start = 0
...     while start != 10:
...         yield start
...         start += 1

>>> gen()
```



# Check Your Understanding: Generators

```
def map_gen(fn, iter1):  
    """  
    >>> i = iter([1, 2, 3, 4])  
    >>> fn = lambda x: x**2  
    >>> m = map_gen(fn, i)  
    >>> next(m)  
    1  
    >>> next(m)  
    4  
    >>> next(m)  
    9  
    >>> next(m)  
    16  
    >>> next(m)  
    Traceback (most recent call last):  
        ...  
    StopIteration  
    """  
    """ *** YOUR CODE HERE *** """
```

# Generators can Yield From Iterators

A `yield from` statement yields all values from an iterable.

```
def a_then_b(a, b):  
    for x in a:  
        yield x  
    for x in b:  
        yield x
```

```
def a_then_b(a, b):  
    yield from a  
    yield from b
```

```
def countdown(k):  
    if k == 0:  
        yield 'Blast off'  
    else:  
        yield k  
        yield from countdown(k-1)
```

# Summary

- We finally made it! What did we even talk about...
- Iterators (bookmarks) are used to iterate over iterables (books).
  - We use the iter method to turn iterables into iterators and we use the next method to get the next element.
- Exceptions can be raised and handled.
- Generators are how we implement iterators in this course and use yield statements.
  - We can use yield from to yield multiple values from an iterable.