STAT606
Computing for Data Science and Statistics

Lecture 8: Functional Programming with `itertools` and `functools`

Functional Programming

In the last lecture, we saw ideas from object oriented programming

- "Everything is an object"
- Every operation is the responsibility of some class/object
- Use side effects to our advantage (e.g., modifying attributes)

In **functional programming**, functions are the central concept, not objects

- "Everything is a function", "data is immutable"
- Avoid side effects at all costs
- Use pure functions (and "meta-functions") as much as possible
- Iterators (or their equivalents) become hugely important

Iterators

An iterator is an object that represents a “data stream”

Supports method `__next__()`:

returns next element of the stream/sequence

raises `StopIteration` error when there are no more elements left

Iterators

An iterator is an object that represents a “data stream”

Supports method `__next__()`:

returns next element of the stream/sequence

raises `StopIteration` error when there are no more elements left

```python
class Squares():
    '''Iterator over the squares.'''
    def __init__(self):
        self.n = 0
    def __next__(self):
        (self.n, k) = (self.n+1, self.n)
        return(k*k)
s = Squares()
[next(s) for _ in range(10)]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

`__next__()` is the important point, here. It returns a value, the next square.

`next(iter)` is equivalent to calling `__next__()`. Variable _ in the list comprehension is a placeholder, tells Python to ignore the value.

Iterators

```
t = [1,2]
titer = iter(t)
next(titer)
```

1

```
next(titer)
```

2

```
next(titer)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-20-105e88283d1e> in <module>()
----> 1 next(titer)

StopIteration:
```

Lists are **not** iterators, but we can turn a list **into** an iterator by calling `iter()` on it. Thus, lists are **iterable**, meaning that it is possible to obtain an iterator over their elements. https://docs.python.org/3/glossary.html#term-iterable

From the documentation: "When an iterable object is passed as an argument to the built-in function iter(), it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call iter() or deal with iterator objects yourself. The for statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop."

Iterators

```
t = [1,2]
titer = iter(t)
next(titer)
```

```
1
```

Lists are **not** iterators, so we first have to turn the list `t` into an iterator using the function `iter()`.

```
next(titer)
```

```
2
```

Now, each time we call `next()`, we get the next element in the list. **Reminder:** `next(iter)` and `iter.__next__()` are equivalent.

```
next(titer)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-20-105e88283d1e> in <module>()
----> 1 next(titer)

StopIteration:
```

Once we run out of elements, we get an error.

Iterators

You are already familiar with iterators from previous lectures. When you ask Python to traverse an object `obj` with a for-loop, Python calls `iter(obj)` to obtain an iterator over the elements of `obj`.

```
t = [1,2,3]
for x in t:
    print(x)
print()
for x in iter(t):
    print(x)
```

```
1
2
3

1
2
3
```

These two for-loops are equivalent. The first one hides the call to `iter()` from you, whereas in the second, we are doing the work that Python would otherwise do for us by casting `t` to an iterator.

Iterators

You are already familiar with iterators from previous lectures. When you ask Python to traverse an object `obj` with a for-loop, Python calls `iter(obj)` to obtain an iterator over the elements of `obj`.

```
t = [1,2,3]
for x in t:
    print(x)
print()
for x in iter(t):
    print(x)
```

These two for-loops are equivalent. The first one hides the call to `iter()` from you, whereas in the second, we are doing the work that Python would otherwise do for us by casting `t` to an iterator.

```
1
2
3

1
2
3
```

A useful note from the documentation: “There is a subtlety when the sequence is being modified by the loop (this can only occur for mutable sequences, i.e. lists). An internal counter is used to keep track of which item is used next, and this is incremented on each iteration. When this counter has reached the length of the sequence the loop terminates. This means that if the suite deletes the current (or a previous) item from the sequence, the next item will be skipped (since it gets the index of the current item which has already been treated). Likewise, if the suite inserts an item in the sequence before the current item, the current item will be treated again the next time through the loop.”

Iterators

If we try to iterate over an object that is not iterable, we're going to get an error.

```
class dummy():
    '''Class that is not iterable,
    because it has neither __next__()
    nor __iter__().'''

d = dummy()
for x in d:
    print(x)
```

Objects of class `dummy` have neither `__iter__()` (i.e., doesn't support `iter()`) nor `__next__()`, so iteration is hopeless. When we try to iterate, Python is going to raise a `TypeError`.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-30-fc084e213893> in <module>()
      5
      6 d = dummy()
----> 7 for x in d:
      8     print(x)

TypeError: 'dummy' object is not iterable
```

Iterators

```
class Squares():
    '''Iterator over the squares.'''
    def __init__(self):
        self.n = 0
    def __next__(self):
        (self.n, k) = (self.n+1, self.n)
        return(k*k)
s = Squares()
for x in s:
    print(x)
```

Merely being an iterator isn't enough, either!
`for X in Y` requires that object `Y` be iterable.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-11-f0187d07bc4c> in <module>()
      7         return(k*k)
      8 s = Squares()
----> 9 for x in s:
     10     print(x)

TypeError: 'Squares' object is not iterable
```

Iterators

Iterable means that an object has the `__iter__()` method, which returns an iterator. So `__iter__()` returns a new object that supports `__next__()`.

```python
class Squares():
    '''Iterator over the squares.'''
    def __init__(self):
        self.n = 0
    def __next__(self):
        (self.n, k) = (self.n+1, self.n)
        return(k*k)
    def __iter__(self):
        return(self)
s = Squares()
for x in s:
    print(x)
```

```
0
1
4
9
16
25
```

Now `Squares` supports `__iter__()` (it just returns itself!), so Python allows us to iterate over it.

This is an infinite loop. Don’t try this at home.

Iterators

```
t1 = ['cat','dog','bird','goat']
t1_iter = iter(t1)
t2 = list(t1_iter)
t1 == t2
```

```
True
```

```
t1 is t2
```

```
False
```

We can turn an iterator *back* into a list, tuple, etc. **Caution:** if you have an iterator like our `Squares` example earlier, this list is infinite and you'll just run out of memory.

Many built-in functions work on iterators. e.g., `max`, `min`, `sum`, work on any iterator (provided elements support the operation); `in` operator will also work on any iterator

Warning: Once again, care must be taken if the iterator is infinite.

List Comprehensions and Generator Expressions

Recall that a list comprehension creates a list from an iterable

```
def square(k):
    return(k*k)
[square(x) for x in range(17) if x%2==0]
```

```
[0, 4, 16, 36, 64, 100, 144, 196, 256]
```

List comprehension computes and returns the whole list. What if the iterable were infinite? Then this list comprehension would never return!

```
s = Squares()
[x**2 for x in s]
```

This list comprehension is going to be infinite! But I really ought to be able to get an iterator over the squares of the elements of `Squares` object `s`...

```
sqgen = (x**2 for x in s)
sqgen
```

```
<generator object <genexpr> at 0x106d02780>
```

This is the motivation for **generator expressions**. Generator expressions are like list comprehensions, but they create an iterator rather than a list.

Generator expressions are written like list comprehensions, but with parentheses instead of square brackets.

Generators

Related to generator expressions are **generators**

Provide a simple way to write iterators (avoids having to create a new class)

```
def harmonic(n):
    return(sum([1/k for k in range(1,n+1)]))
harmonic(10)
```

```
2.9289682539682538
```

Each time we call this function, a local namespace is created, we do a bunch of work there, and then all that work disappears when the namespace is destroyed.

```
def harmonic():
    (h,n) = (0,1)
    while True:
        (h,n) = (h+1/n, n+1)
        yield h
h = harmonic()
[next(h) for _ in range(3)]
```

```
[1.0, 1.5, 1.8333333333333333]
```

Alternatively, we can write `harmonic` as a **generator**. Generators work like functions, but they maintain internal state, and they `yield` instead of `return`. Each time a generator gets called, it runs until it encounters a `yield` statement or reaches the end of the `def` block.

https://en.wikipedia.org/wiki/Harmonic_number

Generators

```
def harmonic():
    (h,n) = (0,1)
    while True:
        (h,n) = (h+1/n, n+1)
        yield h
h = harmonic()
h
```

```
<generator object harmonic at 0x1053b9fc0>
```

```
next(h)
```

```
1.0
```

```
next(h)
```

```
1.5
```

```
next(h)
```

```
1.8333333333333333
```

Python sees the `yield` keyword and determines that this should be a generator definition rather than a function definition.

Generators

```
def harmonic():
    (h,n) = (0,1)
    while True:
        (h,n) = (h+1/n, n+1)
        yield h
h = harmonic()
h
```

```
<generator object harmonic at 0x1053b9fc0>
```

Python sees the `yield` keyword and determines that this should be a generator definition rather than a function definition.

Create a new `harmonic` generator. Inside this object, Python keeps track of where in the `def` code we are. So far, no code has been run.

```
next(h)
```

```
1.0
```

```
next(h)
```

```
1.5
```

```
next(h)
```

```
1.8333333333333333
```

Generators

```
def harmonic():
    (h,n) = (0,1)
    while True:
        (h,n) = (h+1/n, n+1)
        yield h
h = harmonic()
h
```

```
<generator object harmonic at 0x1053b9fc0>
```

Python sees the `yield` keyword and determines that this should be a generator definition rather than a function definition.

```
next(h)
```

```
1.0
```

Each time we call `next`, Python runs the code in h from where it left off until it encounters a `yield` statement.

```
next(h)
```

```
1.5
```

```
next(h)
```

```
1.8333333333333333
```

Generators

```
def harmonic():
    (h,n) = (0,1)
    while True:
        (h,n) = (h+1/n, n+1)
        yield h
h = harmonic()
h
```

```
<generator object harmonic at 0x1053b9fc0>
```

```
next(h)
```

```
1.0
```

```
next(h)
```

```
1.5
```

```
next(h)
```

```
1.8333333333333333
```

Python sees the `yield` keyword and determines that this should be a generator definition rather than a function definition.

Each time we call `next`, Python runs the code in h from where it left off until it encounters a `yield` statement.

Generators

```python
def harmonic():
    (h,n) = (0,1)
    while True:
        (h,n) = (h+1/n, n+1)
        yield h
h = harmonic()
h
```

```
<generator object harmonic at 0x1053b9fc0>
```

```python
next(h)
```

```
1.0
```

```python
next(h)
```

```
1.5
```

```python
next(h)
```

```
1.8333333333333333
```

Python sees the `yield` keyword and determines that this should be a generator definition rather than a function definition.

Each time we call `next`, Python runs the code in h from where it left off until it encounters a `yield` statement.

Generators

```
def harmonic():
    (h,n) = (0,1)
    while True:
        (h,n) = (h+1/n, n+1)
        yield h
h = harmonic()
h
```

```
<generator object harmonic at 0x1053b9fc0>
```

```
next(h)
```

```
1.0
```

```
next(h)
```

```
1.5
```

```
next(h)
```

```
1.8333333333333333
```

Python sees the `yield` keyword and determines that this should be a generator definition rather than a function definition.

Each time we call `next`, Python runs the code in h from where it left off until it encounters a `yield` statement.

Generators

```
def harmonic():
    (h,n) = (0,1)
    while True:
        (h,n) = (h+1/n, n+1)
        yield h
h = harmonic()
h
```

```
<generator object harmonic at 0x1053b9fc0>
```

```
next(h)
```

```
1.0
```

```
next(h)
```

```
1.5
```

```
next(h)
```

```
1.8333333333333333
```

Python sees the `yield` keyword and determines that this should be a generator definition rather than a function definition.

If/when we run out of `yield` statements (i.e., because we reach the end of the definition block), the generator returns a `StopIteration` error, as required of an iterator (not shown here).

Generators

Generators supply a few more bells and whistles

- Ability to pass values *into* the generator to modify behavior
- Can make generators both produce and consume information
    - **Coroutines** as opposed to **subroutines**

See generator documentation for more:

https://docs.python.org/3/reference/expressions.html#generator-iterator-methods

`zip`, revisited

```
h = harmonic()
s = Squares()
z = zip(h,s)
z
```

```
<zip at 0x10608ea48>
```

```
[next(z) for x in range(10)]
```

```
[(1.0, 0),
 (1.5, 1),
 (1.8333333333333333, 4),
 (2.083333333333333, 9),
 (2.283333333333333, 16),
 (2.4499999999999997, 25),
 (2.5928571428571425, 36),
 (2.7178571428571425, 49),
 (2.8289682539682537, 64),
 (2.9289682539682538, 81)]
```

Recall that `zip` takes two or more iterables and returns an iterator over tuples

Here are two infinite iterators, and we `zip` them. So `z` should also be an infinite iterator. But this expression doesn't result in an infinite evaluation...

The trick is that `zip` uses **lazy evaluation**. Rather than trying to build all the tuples right when we call `zip`, Python is lazy. It only builds tuples as we ask for them! We'll see this plenty more in this course. https://en.wikipedia.org/wiki/Lazy_evaluation

Map and Filter

Recall:

- **map** operation applies a function to every element of a sequence
  - Yields a new, transformed sequence
- **filter** operation removes from a sequence all elements failing some condition
  - Again, yields a new, filtered sequence

Map

We saw how to achieve a map operation using list comprehensions

But there's also the Python **map** function:

```
def square_plus1(x):
    return x**2+1
map(square_plus1, range(10))
```

```
<map at 0x102c084e0>
```

```
list(map(square_plus1, range(10)))
```

```
[1, 2, 5, 10, 17, 26, 37, 50, 65, 82]
```

From the documentation:
map(*function*, *iterable*, *...*)
Return an iterator that applies *function* to every item of *iterable*, yielding the results.

`map` and range both produce special kinds of iterators.

Map

```
def poly(x,y):
    return(x*y - 3*x - y)
list(map(poly, range(1,11), range(10,0,-1)))
```

```
[-3, 3, 7, 9, 9, 7, 3, -3, -11, -21]
```

The first argument to `map` is a function; remaining arguments are one or more iterables.

```
list(map(max, [1, 1, 2, 3, 5, 8, 13], range(1,8), 7*[2]))
```

```
[2, 2, 3, 4, 5, 8, 13]
```

```
list(map(poly, range(10), range(10), range(10)))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-247-150a3296c401> in <module>()
----> 1 list(map(poly, range(10), range(10), range(10)))

TypeError: poly() takes 2 positional arguments but 3 were given
```

Number of iterables and number of function arguments must agree!

Aside: `lambda` expressions

Lambda expressions let you define functions without using a `def` statement

- Called an **in-line function** or **anonymous function**
- Name is a reference to *lambda calculus*, a concept from symbolic logic

```
def my_square(x):
    return x**2
list(map(my_square, range(1,10)))
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Define a function, then pass it to `map`.

```
list(map(lambda x: x**2, range(1,10)))
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Alternatively, define an equivalent function **in-line**, using a **lambda statement**.

A lambda expression returns a function, so `my_square` and `lambda x: x**2` are, in a certain sense, equivalent.

Aside: `lambda` expressions

```
lambda x : x**2 + 1
```

```
<function __main__.<lambda>>
```

Arguments of the function are listed before the colon. So this function takes a single argument...

```
lambda x,y,z,n : x**n + y**n == z**n
```

```
<function __main__.<lambda>>
```

...while this one takes four.

```
(lambda x,y,z,n : x**n + y**n == z**n)(3,4,5,2)
```

```
True
```

```
(lambda x,y,z,n : x**n + y**n == z**n)(13,17,19,42)
```

```
False
```

```
my_square
```

```
<function __main__.my_square>
```

Aside: `lambda` expressions

```
1  lambda x : x**2 + 1
```

```
<function __main__.<lambda>>
```

Return value of the function is listed on the right of the colon. So this function returns the square of its input plus 1....

```
1  lambda x,y,z,n : x**n + y**n == z**n
```

```
<function __main__.<lambda>>
```

...and this one returns a Boolean stating whether or not the four numbers satisfy Fermat's last theorem.

```
1  (lambda x,y,z,n : x**n + y**n == z**n)(3,4,5,2)
```

```
True
```

```
1  (lambda x,y,z,n : x**n + y**n == z**n)(13,17,19,42)
```

```
False
```

```
1  my_square
```

```
<function __main__.my_square>
```

https://en.wikipedia.org/wiki/Fermat's_Last_Theorem

Aside: `lambda` expressions

```
lambda x : x**2 + 1
```

```
<function __main__.<lambda>>
```

```
lambda x,y,z,n : x**n + y**n == z**n
```

```
<function __main__.<lambda>>
```

Lambda expressions return actual functions, which we can apply to inputs.

```
(lambda x,y,z,n : x**n + y**n == z**n)(3,4,5,2)
```

```
True
```

```
(lambda x,y,z,n : x**n + y**n == z**n)(13,17,19,42)
```

```
False
```

```
my_square
```

```
<function __main__.my_square>
```

Function names are stored in an attribute `__name__`. Since lambda expressions yield anonymous functions, they all have the generic name `'<lambda>'`.

Aside: `lambda` expressions

```
1 f = lambda x : x+'goat'
2 f('cat')
```

```
'catgoat'
```

```
1 (lambda x : 2*x)(21)
```

```
42
```

```
1 list(map(lambda x: x**2, range(1,10)))
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Lambda expressions can be used anywhere you would use a function. Note that the term **anonymous function** makes sense: the lambda expression defines a function, but it never gets a variable name (unless we assign it to something, like in the `'goat'` example to the left).

First-class functions

```
f = lambda x : x+'goat'
f('cat')
```

```
'catgoat'
```

```
def my_square(x):
    return(x**2)
my_square
```

```
<function __main__.my_square>
```

The fact that we can have variables whose values are functions is actually quite special. We say that Python has **first-class functions**. That is, functions are perfectly reasonable values for a variable to have.

You've seen these ideas before if you've used R's `tapply` (or similar), MATLAB's function handles, C/C++ function pointers, etc.

Filter

The list filter expression also has an analogous function, `filter`.

```
fibo = [1,1,2,3,5,8,13]
def is_even(x):
    return(x%2==0)
filter(is_even, fibo)
```

`filter` takes a Boolean function and an iterator and returns an iterator of only the elements that evaluated to `True`.

```
<filter at 0x10223ef28>
```

Returns its own special iterator.

```
list(filter(is_even, fibo))
```

```
[2, 8]
```

Second argument to `filter` (and `map`) can be **any** iterator. Here we are filtering a generator.

```
list(filter(is_even, (x**2 for x in range(10))))
```

```
[0, 4, 16, 36, 64]
```

Filter

```
list(filter(lambda x: x%2==0, range(10)) )
```

```
[0, 2, 4, 6, 8]
```

It's often more convenient to just use a lambda expression in-line instead of defining a Boolean function elsewhere.

```
list(filter(is_even, range(10)))
```

```
[0, 2, 4, 6, 8]
```

```
list(filter(lambda t: t[0]**2 + t[1]**2 == t[2]**2, [(3,3,3),(3,4,5),(4,5,6),(5,12,13)]))
```

```
[(3, 4, 5), (5, 12, 13)]
```

Lambda expressions don't support scatter/gather, so you have to use this kind of pattern to process tuples. Worry not! Another Python module does support this, and we'll see it in a few slides.

What about reduce?

We saw `map` and `filter` earlier, but we can't have MapReduce without **reduce**

```
sum(range(1,11))
```

```
55
```

Reduce operations **reduce** an iterator (i.e., a sequence) to a single element. Sum is a good example of a reduce function.

```
import functools
functools.reduce(lambda x,y: x+y, range(1,11))
```

```
55
```

`functools` contains a bunch of useful functional programming functions, including reduce.

`functools.reduce` takes a function and and iterator and performs a **reduce** operation on the iterator using the function.

A reduce operation takes a function and a sequence, and returns a single object (typically of the same type as the elements of the sequence). `sum()` is a good example of a reduce operation, but it's hardly the only one.

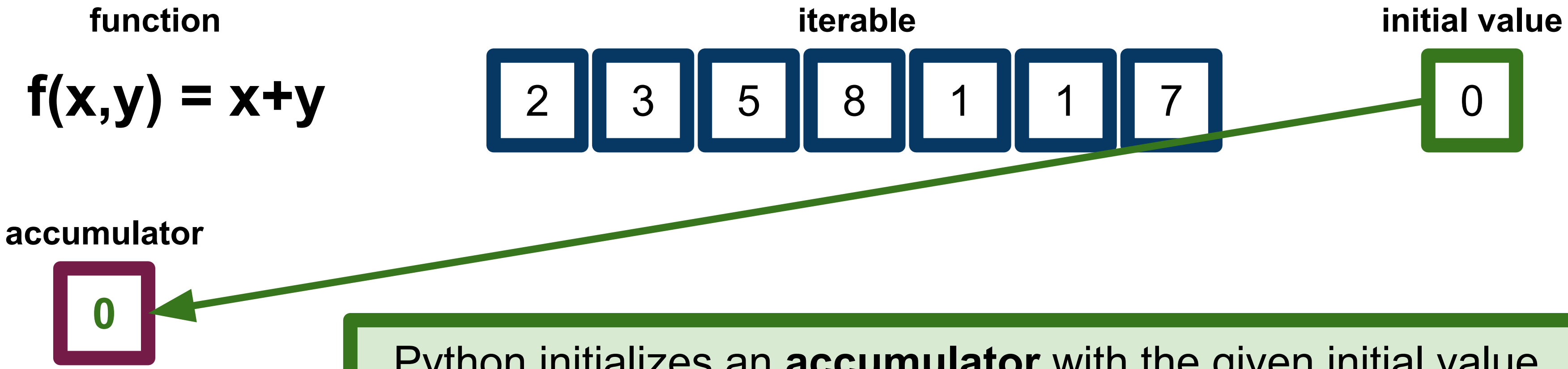
Reduce operations

Three fundamental pieces:

function	**iterable**	**initial value**
$f(x,y) = x+y$	2 3 5 8 1 1 7	0

Reduce operations

Three fundamental pieces:



Python initializes an **accumulator** with the given initial value.
Think of the accumulator as a “running total”.

Reduce operations

Three fundamental pieces:

function

f(x,y) = x+y

iterable

2	3	5	8	1	1	7

initial value

0

accumulator

0

Now, Python repeatedly updates the accumulator, with
`accumulator = f(accumulator,y)`
where `y` traverses the sequence

Reduce operations

Three fundamental pieces:

function

$$f(x,y) = x+y$$

iterable

2	3	5	8	1	1	7

initial value

0

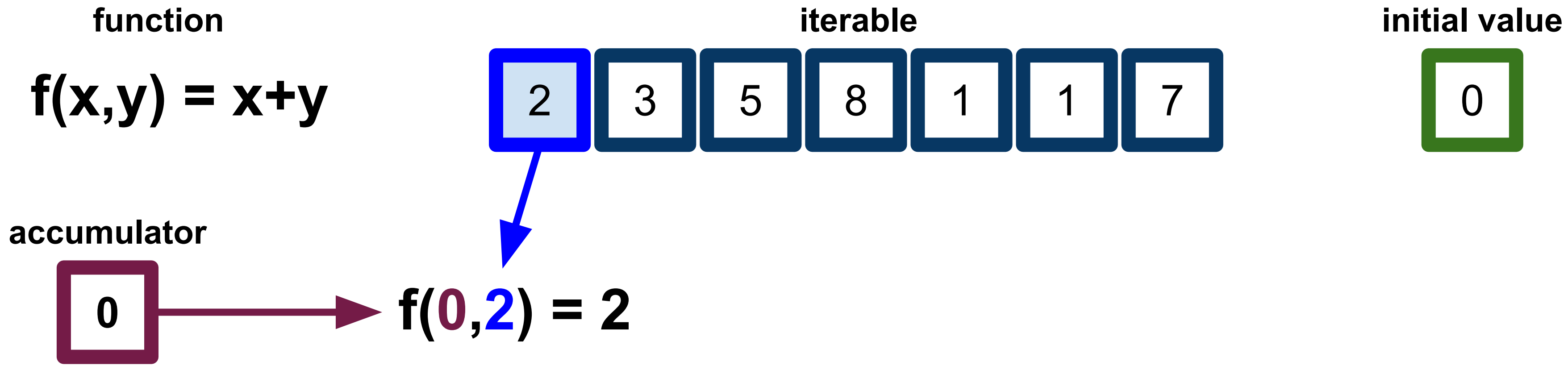
accumulator

0

Now, Python repeatedly updates the accumulator, with
`accumulator = f(accumulator,y)`
where `y` traverses the sequence

Reduce operations

Three fundamental pieces:



Now, Python repeatedly updates the accumulator, with

`accumulator = f(accumulator,y)`

where `y` traverses the sequence

Reduce operations

Three fundamental pieces:

function

f(x,y) = x+y

iterable

2	3	5	8	1	1	7

initial value

0

accumulator

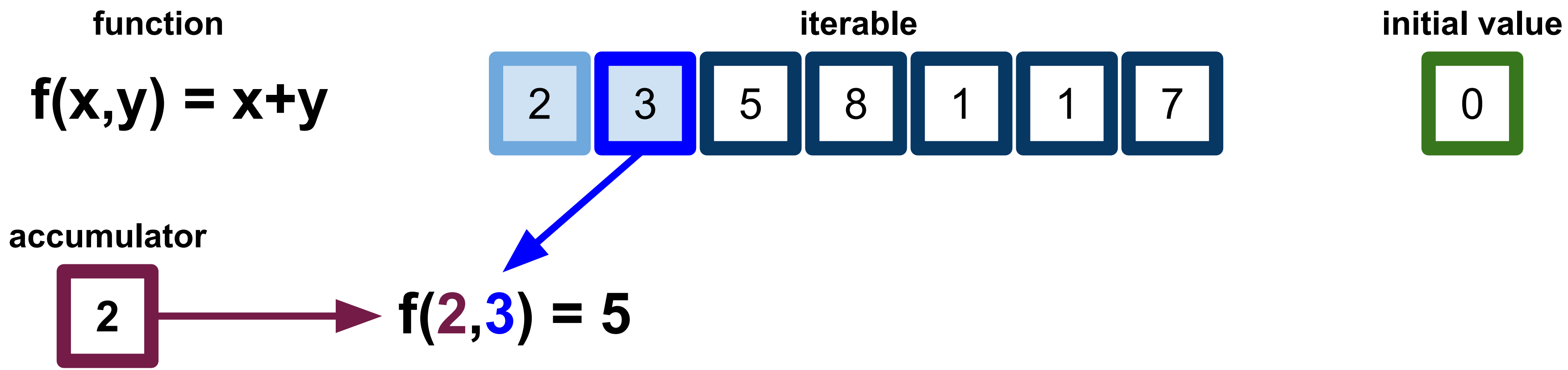
2

f(0,2) = 2

Now, Python repeatedly updates the accumulator, with
`accumulator = f(accumulator,y)`
where `y` traverses the sequence

Reduce operations

Three fundamental pieces:



Now, Python repeatedly updates the accumulator, with
`accumulator = f(accumulator,y)`
where `y` traverses the sequence

Reduce operations

Three fundamental pieces:

function

f(x,y) = x+y

iterable

2	3	5	8	1	1	7

initial value

0

accumulator

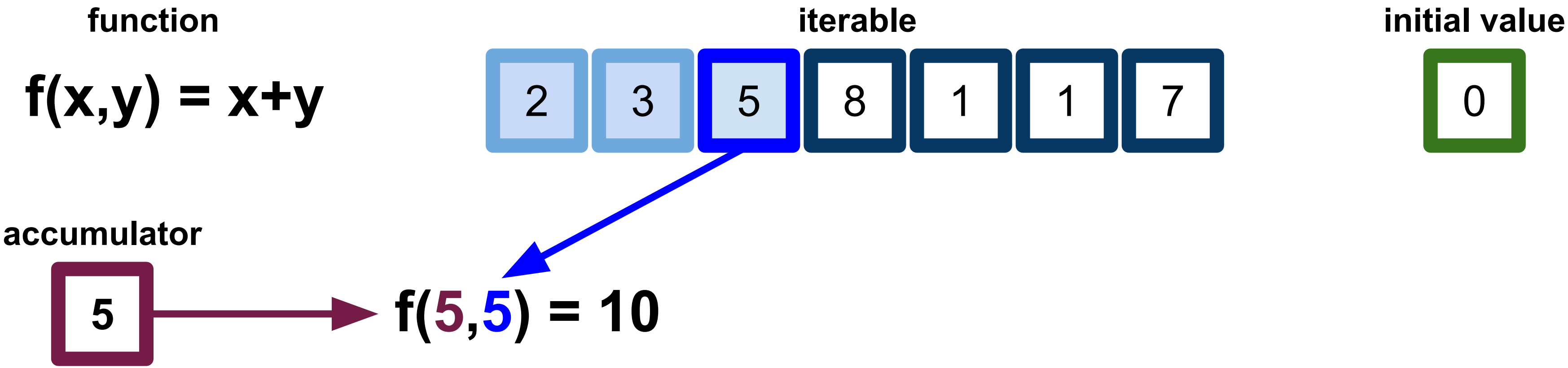
5

f(2,3) = 5

Now, Python repeatedly updates the accumulator, with
`accumulator = f(accumulator,y)`
where `y` traverses the sequence

Reduce operations

Three fundamental pieces:



Now, Python repeatedly updates the accumulator, with
`accumulator = f(accumulator,y)`
where `y` traverses the sequence

Reduce operations

Three fundamental pieces:

function	**iterable**	**initial value**
$f(x,y) = x+y$	2 3 5 8 1 1 7	0

accumulator

10 ← **$f(5,5) = 10$**

Now, Python repeatedly updates the accumulator, with

`accumulator = f(accumulator,y)`

where `y` traverses the sequence

Reduce operations

Three fundamental pieces:

function

$$f(x,y) = x+y$$

iterable

2	3	5	8	1	1	7

initial value

0

accumulator

10

...and so on.

Reduce operations

Three fundamental pieces:

function

$$f(x,y) = x+y$$

iterable

2	3	5	8	1	1	7

initial value

0

accumulator

27

Once Python gets a `StopIteration` error indicating that the iterator has no more elements, it returns the value in the accumulator.

Reduce operations

Three fundamental pieces:

function

$$f(x,y) = x+y$$

iterable

2	3	5	8	1	1	7

initial value

0

```
functools.reduce(lambda x,y:x+y, range(10), 0)
```

```
45
```

```
functools.reduce(lambda x,y:x+y, range(10))
```

```
45
```

```
functools.reduce(lambda x,y:x+y, [2.71828])
```

```
2.71828
```

If the initial value isn't supplied, Python initializes the accumulator as `acc = f(x,y)` where `x` and `y` are the first two elements of the iterator. If the iterator is length 1, it just returns that element. All told, it's best to always specify the initial value, except in very simple cases (like these slides).

Reduce operations

Three fundamental pieces:

function

f(x,y) = x+y

iterable

2 3 5 8 1 1 7

initial value

0

```
functools.reduce(lambda x,y:x+y,[])
```

Warning: if the iterator supplied to `reduce` is empty, then we really do need the initial value!

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-116-a9df9cc43559> in <module>()
----> 1 functools.reduce(lambda x,y:x+y,[])

TypeError: reduce() of empty sequence with no initial value
```

Reduce in Python

`reduce` is not included as a built-in function in Python, unlike `map` and `filter`

Because developers felt that reduce is not "Pythonic"

The argument is that reduce operations can always be written as a for-loop:

```
import functools
functools.reduce(lambda x,y: x+y, range(10))
```

```
45
```

```
acc = 0
for i in range(10):
    acc += i
acc
```

```
45
```

Reduce in Python

`reduce` is not included as a built-in function in Python, unlike `map` and `filter`

Because developers felt that reduce is not “Pythonic”

The argument is that reduce operations can always be written as a for-loop:

```
import functools
functools.reduce(lambda x,y: x+y, range(10))
```

```
45
```

```
acc = 0
for i in range(10):
    acc += i
acc
```

```
45
```

This criticism is mostly correct, but we’ll see later in the course when we cover MapReduce that there are cases where we really do want a proper reduce function.

Reduce in Python

```
import random
numbers = [random.randint(1,1000) for _ in range(100)]
functools.reduce(lambda x,y:x if x > y else y, numbers)
```

```
989
```

All of the standard reduce-like functions are easily reimplemented with reduce statements, like this example, with `max`. Note the use of Python's in-line conditional statement.

```
def tuple_add(x,y):
    if len(x) != len(y):
        raise TypeError('Tuple lengths must agree')
    r = len(x)*[0]
    for i in range(len(x)):
        r[i] = x[i] + y[i]
    return tuple(r)
functools.reduce(tuple_add, [(1,2),(1,3),(2,5),(3,7),(5,11)])
```

```
(12, 28)
```

More often, one has to implement the pairwise function. For example, here we have implemented a function for entrywise addition of tuples.

Note: there are "more functional" ways to do this. Since tuples are themselves iterable, we could write a clever "function of functions" to do this more gracefully. More on this soon.

Quantifiers over iterables: `any()` and `all()`

```
1 any([False,True,False])
```

True

```
1 any((0,'',0.0))
```

False

```
1 all([(1,0),1,'cat'])
```

True

```
1 all(map(is_even,fibo))
```

False

`any` takes an iterable as its input and returns `True` if and only if one or more elements is `True`.

Reminder: 0, 0.0, empty string, empty list, etc all evaluate to `False`. Just about everything else evaluates to `True`.

`all` takes an iterable as its input and returns `True` if and only if all elements are `True`.

Quantifiers over iterables: `any()` and `all()`

```
def is_prime(n):
    return not any((n%x==0 for x in range(2,n)))
is_prime(8675309)
```

```
True
```

```
is_prime(8675310)
```

```
False
```

Here's a nice example of why functional programming is useful. Complicated functions become elegant one-liners!

Of course, sometimes that elegance comes at the cost of efficiency. In this example, we're failing to use a speedup that would be gained from using, e.g., the sieve of Eratosthenes and stopping checking above `sqrt(n)`.

https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

`any` and `all` are lazy

```
1 any([False,True,False])
```

```
True
```

```
1 any((0,'',0.0))
```

```
False
```

```
1 all([(1,0),1,'cat'])
```

```
True
```

```
1 all(map(is_even,fibo))
```

```
False
```

As soon as `any` finds a `True` element, it returns `True`. As soon as `all` finds a `False` element, it returns `False`. This is a simpler (i.e., less general) notion of laziness than lazy evaluation, but the underlying motivation is the same. Do as little work as is necessary to get your answer!

Related: `itertools.accumulate`

```
1 itertools.accumulate(range(1,10), lambda x,y:x+y)
```

```
<itertools.accumulate at 0x10a6aa348>
```

`itertools.accumulate` performs a reduce operation, but it returns an iterator over the partial "sums" of its argument. Returns an empty iterator if argument is empty.

```
1 list(itertools.accumulate(range(1,10), lambda x,y:x+y))
```

```
[1, 3, 6, 10, 15, 21, 28, 36, 45]
```

```
1 list(itertools.accumulate([(1,2),(1,3),(2,5),(3,7),(5,11)], tuple_add))
```

```
[(1, 2), (2, 5), (4, 10), (7, 17), (12, 28)]
```

I put "sums" in quotes above, because of course the function need not be addition. The point is that we get an iterator over the values of the accumulator at each step of the reduce operation.

Working with iterators: `itertools`

```
import itertools
sevens = itertools.count(7,7)
[next(sevens) for x in range(10)]
```

```
[7, 14, 21, 28, 35, 42, 49, 56, 63, 70]
```

`itertools.count(x,y)` returns an infinite iterator of numbers starting at `x` and proceeding in increments of `y`.

```
list(itertools.accumulate(range(10)))
```

```
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
```

`itertools.accumulate(t)` returns an iterator of partial sums of `t`. Or partial "sums" if we specify a different function.

```
list(itertools.accumulate(range(1,10),max))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
list(itertools.filterfalse(is_even, fibo))
```

```
[1, 1, 3, 5, 13]
```

`itertools.filterfalse(t)` is like the opposite of `filter`.

```
list(itertools.starmap(poly,[(1,1),(1,2),(2,1),(3,4)]))
```

```
[-3, -3, -5, -1]
```

`itertools.starmap` similar to `map`, but applies multi-argument function to tuples. Name is reference to the `*args` notation.

https://docs.python.org/3/library/itertools.html#module-itertools

More `itertools`: combinations

```
list(itertools.combinations([1,2,3,4], 2))
```

```
[(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
```

```
list(itertools.permutations([1,2,3], 2))
```

```
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
```

```
list(itertools.combinations_with_replacement([1,2,3,4], 2))
```

```
[(1, 1),
 (1, 2),
 (1, 3),
 (1, 4),
 (2, 2),
 (2, 3),
 (2, 4),
 (3, 3),
 (3, 4),
 (4, 4)]
```

`itertools` also includes some combinatorial functions that can be useful on occasion.

Aside: Python `operator` module

It's awfully annoying to have to write `lambda x,y:x+y` all the time

```
functools.reduce(lambda x,y:x*y, range(1,10))
```

```
362880
```

Here is what we'd *like* to write, but of course it's a syntax error.

```
functools.reduce(*,range(1,10))
```

```
  File "<ipython-input-90-461403074121>", line 1
    functools.reduce(*,range(1,10))
                      ^
SyntaxError: invalid syntax
```

`operator.mul` gives us `*`, but as a function, just as though we wrote a lambda expression.

```
import operator
functools.reduce(operator.mul,range(1,10))
```

```
362880
```

`operator` includes many other functions:

Math: `add()`, `sub()`, `mul()`, `abs()`, etc.

Logic: `not_()`, `truth()`.

Bitwise: `and_()`, `or_()`, `invert()`.

Comparison: `eq()`, `ne()`, `lt()`, `le()`, etc.

Identity: `is_()`, `is_not()`.

https://docs.python.org/3/library/operator.html#module-operator

More functional patterns: `functools`

`functools` module provides a number of functional programming constructions

```
import functools
pow2 = functools.partial(math.pow, 2)
pow2
```

```
functools.partial(<built-in function pow>, 2)
```

`functools.partial` takes a function and a set of arguments to pass to the function. Returns a function with some of its arguments “fixed”.

```
list(map( pow2, range(10) ))
```

```
[1.0, 2.0, 4.0, 8.0, 16.0, 32.0, 64.0, 128.0, 256.0, 512.0]
```

So in this case, it’s like we got a new function, `pow2(x) == math.pow(2,x)`

```
def my_pow(x=1, y=1):
    return math.pow(x,y)
my_square = functools.partial(my_pow, y=2)
list(map( my_square, range(10) ))
```

```
[0.0, 1.0, 4.0, 9.0, 16.0, 25.0, 36.0, 49.0, 64.0, 81.0]
```

`functools.partial` also lets us pass keyword arguments.

Higher-order functions and currying

`functools.partial` takes a function (and other stuff), returns a function

Called a **higher-order function**

In most other languages, Python's `functools.partial` is called **currying**

```
f = lambda x,y,z : x*y*z
curry1 = functools.partial(f,2)
curry2 = functools.partial(curry1,3)
curry2(4)
```

```
24
```

`curry1` takes two arguments, returns their product times 2.

`curry2` takes one argument `z`, returns `2*3*z` (reminder: `partial` fills positional arguments in order).

```
par = functools.partial(f,2,3)
par(4)
```

```
24
```

Equivalently, just pass both arguments in one call to partial.

Currying is named after logician Haskell Curry
https://en.wikipedia.org/wiki/Currying

Pure functions, again

Recall that a **pure function** is a function that did not have any side effects

Pure functions are especially important in functional programming

- A pure function is really a function (in the mathematical sense)
- Given the same input, it always produces the same output
  - (And doesn’t change the state of our program!)

```
a = 0
def increment_mod():
    global a
    a += 1
def increment_pure(x):
    return x+1
```

This function is a modifier.
It has side effects.

This is a pure function.

Pure functions, again

Recall that a **pure function** was a function that did not have any side effects

Pure functions are especially important in functional programming

A pure function is really a function (in the mathematical sense)

Given the same input, it always produces the same output

(And doesn't change the state of our program!)

```
a = 0
def increment_mod():
    global a
    a += 1
def increment_pure(x):
    return x+1
```

Pure functions are also crucial to having **immutable data**. Think about processing the observations in a data set. We don't want to change the original data file in the process of our analysis! We want to be able to write a pipeline, in which we pass data from one function to another, producing a transformed version of the data at each step.

Pure functions and higher-order functions

Pure functions arise frequently in map/reduce frameworks

A good example of a higher-order function: `compose` takes some functions and produces a new function.

Returning a function is okay, because Python has first-class functions.

You can see why we prefer pure functions for this. If `f` and/or `g` had side effects, this would be a big mess!

```
def compose(*funcs):
    '''Return a new function that is the
    composition of the argument functions.'''
    def inner(data, funcs=funcs):
        result = data
        for f in reversed(funcs):
            result = f(result)
        return result
    return inner
f = lambda x: x**2
g = lambda x: x+1
# compose(g,f) == g(f(x)) == x**2 + 1
list(map(compose(g,f), range(10)))
```

```
[1, 2, 5, 10, 17, 26, 37, 50, 65, 82]
```

Example credit: D. Mertz, *Functional Programming in Python*

Functional vs Object-oriented Programming

```
class LetterCounter():
    '''Counts letters in a text stream'''
    def __init__(self, letter):
        self.letter=letter
        self.count=0
    def increment(self):
        self.count+=1
    def process_file(self, filename):
        with open(filename, 'r') as f:
            for line in f:
                self.process_line(line)
    def process_line(self, line):
        for x in line:
            if x==self.letter:
                self.increment()
    def get_count(self):
        return self.count
lc = LetterCounter('e')
fname = '/Users/keith/Downloads/mobydick.txt'
lc.process_file(fname)
lc.get_count()
```

```
118501
```

Of course, I'm exaggerating the complexity of this object here, but this really is what object-oriented code ends up looking like in the wild.

Contrast that with the simplicity of this functional version of the same letter-counting operation.

```
def count_letter(fname, letter):
    with open(fname, 'r') as f:
        return(sum([c==letter
                    for line in f
                    for w in line
                    for c in w]))
count_letter(fname, 'e')
```

```
118501
```

Why use functional programming?

Some problems are especially well-suited to this paradigm

Example: quicksort

```
def quicksort(t):
    if len(t) <= 1:
        return t # list is already sorted.
    else:
        pivot = t[0]
        return(quicksort([x for x in t if x < pivot]) +
                  [x for x in t if x==pivot] +
                 quicksort([x for x in t if x > pivot]))
quicksort([3,5,4,2,1,6,5,7,4,0,0,2,4])
```

```
[0, 0, 1, 2, 2, 3, 4, 4, 4, 5, 5, 6, 7]
```

See the quicksort Wikipedia page for examples of what this looks like when written in a non-functional style.

https://en.wikipedia.org/wiki/Quicksort

A note on recursion in Python: tail call optimization

M. R. Cook, *A Practical Introduction to Functional Programming*:
“Tail call optimisation is a programming language feature. Each time a function recurses, a new stack frame is created. A stack frame is used to store the arguments and local values for the current function invocation. If a function recurses a large number of times, it is possible for the interpreter or compiler to run out of memory. Languages with tail call optimisation reuse the same stack frame for their entire sequence of recursive calls. Languages like Python that do not have tail call optimisation generally limit the number of times a function may recurse to some number in the thousands.”

Python doesn’t have tail call recursion, so some functional programing patterns simply aren’t well-suited if we may encounter many thousands of layers of recursion. Recall our memoized function for computing the Fibonacci numbers.

Declarative Programming

Describe what the program **should do**, rather than how it does it

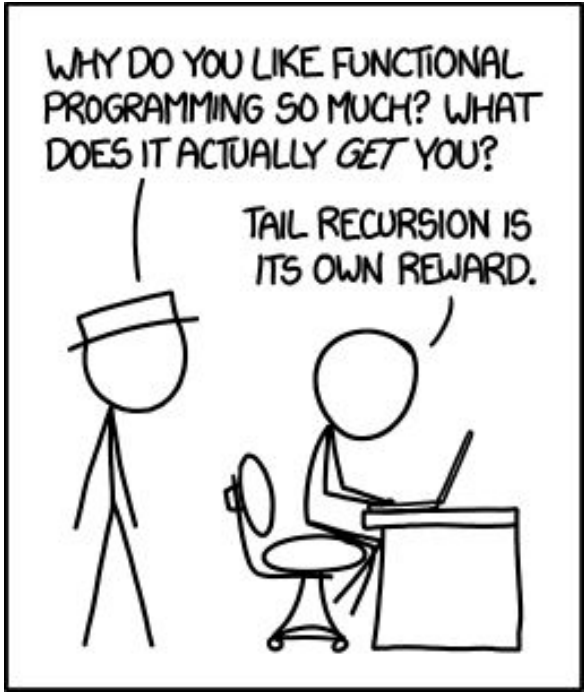
- Implementation details are left up to the language as much as possible

Contrast with **imperative/procedural programming**

- Sequence of statements describes **how** program should proceed
- Most programming you have done in the past is procedural
- Program consists of subroutines that get called, change state of program

Don't worry too much about these distinctions. Most languages are a mix of paradigms, and no single approach is a silver bullet.
Different applications call for different programming paradigms.

Congratulations! You know enough functional programming to get the joke in this xkcd comic!



Alt-text: Functional programming combines the flexibility and power of abstract mathematics with the intuitive clarity of abstract mathematics.

https://xkcd.com/1270/