CS 61A
DISCUSSION NINE

November 10, 2016

TOPICS FOR TODAY

Iterators, generators, streams (, ...turtle graphics demo !?)

Overview

- Iterators are objects that sweep over a collection of items (in a specific order / step size). This happens via repeated application of the `next` method, which is necessarily defined on all iterators.

- Streams are lazily evaluated linked lists. Elements in the list aren’t computed until you specifically ask for them, which means you can represent infinite sequences as streams.

ANNOUNCEMENTS

- No in-person lab on Thanksgiving week (instead, we’ll have a “long, optional” one to be completed on your own)

ITERATORS

ITERATION (REVIEW)

Iterators step through a collection, item by item, via `next`
Iterables “are” the collection, and provide **iterators** via `iter`

If you have an iter**able**, you can get an iter**ator** over it by calling `iter`. Then you can observe all of its elements by repeatedly calling `next` on the iter**ator**.

Be warned: iterators are single-use only! (Once an iterator has gone through all the elements, it’s done; calling `next` on it will give `StopIteration` errors forever.) To get a fresh iterator, one would have to call `iter` on the iterable again.

- Note that depending on the implementation, the second iterator might be finished anyway (e.g. if every iterator modifies the same state variables)

ITERATION (REVIEW)

```
iter(iterable) → iterator
next(iterator) → value, or a StopIteration error
```

ITERATION (REVIEW)

Lots of **built-in** functions take or produce iterators! Be aware of these.

- `map(function, iterable)`
  - Returns an **iterator** over mapped elements in the iterable
- `filter(function, iterable)`
  - Returns an **iterator** over filtered elements from the iterable
- `zip(*iterables)`
  - Returns an **iterator** over aggregations of elements from each of the iterables

ITERATION (REVIEW)

To create an iterable, you could write a class that implements `__iter__` as a generator function.

```
class Primes:
    def __init__(self, n):
        self.n = n # upper limit
    def __iter__(self):
        P = [True for i in range(2, self.n + 1)]
        for i in range(2, self.n + 1):
            if not P[i - 2]: continue
            yield i
            if i > sqrt(self.n): continue
            for j in [i*i + k*i for k in range((self.n - i*i)//i + 1)]:
                P[j - 2] = False

list(Primes(30)) → [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

ITERATION (REVIEW)

Behind the scenes, for-loops really just create iterators using `iter` and then call `next` a bunch of times on the iterator.

```
for x in <expr>:
    <do stuff>
```

– *is equivalent to* –

```
iterator = iter(<expr>)
try:
    while True:
        n = next(iterator)
        <do stuff>
except StopIteration:
    pass
```

ITERATORS E1

```
>>> t = iter([0, 1, 2]) # t is now an iterator
>>> u = iter(t) # does it error...?
```

What happens when you call `iter` on an iterator?

ITERATORS E1

```
>>> t = iter([0, 1, 2]) # t is now an iterator
>>> u = iter(t) # does it error…?
```

What happens when you call `iter` on an iterator?

You get the same iterator back. So… technically speaking, iterators also implement the iterable interface! (Unless they're user-defined iterators that didn't implement the `__iter__` method, but you don't need to worry about that this semester…)

```
>>> t == u
True
>>> t == iter(t)
True
```

ITERATORS E2

Let's implement the `list` function. Here we have the function specification:

```
def list(iterable):
    """Creates a list.

    >>> list(range(4))
    [0, 1, 2, 3]
    >>> list(iter(range(4)))
    [0, 1, 2, 3]
    """
    # YOUR CODE HERE
```

ITERATORS E2

Let's implement the `list` function. Here we have the function specification:

```
def list(iterable):
    iterator = iter(iterable)
    result = []
    try:
        while True:
            result.append(next(iterator))
    except StopIteration:
        return result
```

GENERATORS

GENERATORS (REVIEW)

Generator functions are functions containing `yield` statements.

- These functions return generators when called.

Generators are **iterators** obtained by calling a generator function.

- Every time you call `next` on a generator, it goes through the generator function body until it hits a `yield` – at which point it yields the specified value. The state of the function with respect to this generator is saved, so whenever `next` is called again on the generator, execution of the function body will continue from where it left off.

To create a fresh iterator, just call the generator function again.

GENERATORS (REVIEW)

Here we have a function that returns an iterator over the natural numbers:

```
def gen_naturals():
    curr = 0
    while True:
        yield curr
        curr += 1

>>> gen = gen_naturals()
>>> gen
<generator object gen at ...>
>>> next(gen)
0
>>> next(gen)
1
```

GENERATORS (REVIEW)

`yield from` is like yield, except it yields **all values** from an **iterable**.

```
yield from <expr>
```

– is equivalent to –

```
for x in <expr>:
    yield x
```

GENERATORS (REVIEW)

```
square = lambda x: x * x
def many_squares(s):
    for x in s:
        yield square(x)
    yield from [square(x) for x in s]
    yield from map(square, s)

>>> list(many_squares([1, 2, 3]))
[1, 4, 9, 1, 4, 9, 1, 4, 9]
```

GENERATORS E1

```
>>> def gen_y():
...     yield (7, 8, 9)
...
>>> def gen_yf():
...     yield from (7, 8, 9)
...
>>> next(gen_y())


>>> next(gen_yf())
```

GENERATORS E1

```
>>> def gen_y():
...     yield (7, 8, 9)
...
>>> def gen_yf():
...     yield from (7, 8, 9)
...
>>> next(gen_y())
(7, 8, 9)

>>> next(gen_yf())
7
```

GENERATORS E2

```
>>> def countup():
...     yield from [1, 2, 3]
...     return 4
...     yield from [5, 6]
...
>>> list(countup)
```

What would Python print?

GENERATORS E2

```
>>> def countup():
...     yield from [1, 2, 3]
...     return 4
...     yield from [5, 6]
...
>>> list(countup)
```

What would Python print?

Trick question, `countup` is just a generator function (NOT a generator; NOT an iterable)

`(TypeError: ‘function’ object is not iterable)`

GENERATORS E3

```
>>> def countup():
...     yield from [1, 2, 3]
...     return 4
...     yield from [5, 6]
...
>>> list(countup())
```

What would Python print?

GENERATORS E3

```
>>> def countup():
...     yield from [1, 2, 3]
...     return 4
...     yield from [5, 6]
...
>>> list(countup())
```

What would Python print?

```
[1, 2, 3]
```

Return results in a `StopIteration` (incl. the implicit return at the end of the function). The value returned is not actually returned by the generator function.

GENERATORS E4

```
>>> def myst(iterable, n):
...     if n <= 0: return
...     iterator = iter(s0)
...     yield next(iterator)
...     if n % 2:
...         yield from myst(iterable, n - 1)
...     else:
...         yield from myst(iterator, n - 1)
...
>>> gen = lambda: (yield from range(7))
>>> [e for e in myst(gen(), 3)]
```

What would Python print?

GENERATORS E4

```
>>> def myst(iterable, n):
...     if n <= 0: return
...     iterator = iter(s0)
...     yield next(iterator)
...     if n % 2:
...         yield from myst(iterable, n - 1)
...     else:
...         yield from myst(iterator, n - 1)
...
>>> gen = lambda: (yield from range(7))
>>> [e for e in myst(gen(), 3)]
```

`[0, 1, 2].` Note that `` `yield from myst(iterable, n - 1)` `` is equivalent to `` `yield from myst(iterator, n - 1)` `` in this case. (Why?)

GENERATORS E5

```
>>> def myst(iterable, n):
...     if n <= 0: return
...     iterator = iter(s0)
...     yield next(iterator)
...     if n % 2:
...         yield from myst(iterable, n - 1)
...     else:
...         yield from myst(iterator, n - 1)
...
>>> [e for e in myst(range(7), 3)]
```

Now what would Python print?
(This is exactly the same code as last time, except we're calling `myst` on a different iterable.)

GENERATORS E5

```
>>> def myst(iterable, n):
...     if n <= 0: return
...     iterator = iter(s0)
...     yield next(iterator)
...     if n % 2:
...         yield from myst(iterable, n - 1)
...     else:
...         yield from myst(iterator, n - 1)
...
>>> [e for e in myst(range(7), 3)]
```

`[0, 0, 1]`. Why? Because `iter` is returning a fresh iterator every time! (In the last case, it wasn't – because the iterable that `iter` was being called on was an iterator. *Note, again: iterators are technically iterables because you can call* `iter` *on them.*)

GENERATORS E6

```
>>> def gen_e6(n):
...     yield n / (n - 1)
...     yield from gen_e6(n - 1)
...
>>> e6 = gen_e6(3)
>>> next(e6)


>>> next(e6)


>>> next(e6)


>>> next(e6)
```

GENERATORS E6

```
>>> def gen_e6(n):
...     yield n / (n - 1)
...     yield from gen_e6(n - 1)
...
>>> e6 = gen_e6(3)
>>> next(e6)
1.5

>>> next(e6)
2.0

>>> next(e6)
ZeroDivisionError: division by zero

>>> next(e6)
StopIteration
```

GENERATORS 1.5.1

Write a generator function that combines the elements of two input iterators using a given combiner function. When either iterator runs out of elements, the whole generator should also run out of elements.

```
def combiner(iterator1, iterator2, combiner):
    # YOUR-CODE-HERE

>>> from operator import add
>>> evens = combiner(gen_naturals(), gen_naturals(), add)
>>> next(evens)
0
>>> next(evens)
2
>>> next(evens)
4
```

GENERATORS 1.5.1/2

```
def combiner(iterator1, iterator2, combiner):
    while True:
        yield combiner(next(iterator1), next(iterator2))
```

WWPP?

```
>>> nats = gen_naturals()
>>> doubled_nats = combiner(nats, nats, add)
>>> next(doubled_nats)

>>> next(doubled_nats)
```

GENERATORS 1.5.1/2

```
def combiner(iterator1, iterator2, combiner):
    while True:
        yield combiner(next(iterator1), next(iterator2))
```

WWPP?

```
>>> nats = gen_naturals()
>>> doubled_nats = combiner(nats, nats, add)
>>> next(doubled_nats)
1
>>> next(doubled_nats)
5
```

GENERATORS 1.5.3

Write a generator function that goes through all subsets of the positive integers from 1 to `n`. Each call to this generator's `next` method will return a list of subsets of the set `[1, 2, …, n]`, where `n` is the number of times `next` was previously called.

```
def generate_subsets():
    """
    >>> subsets = generate_subsets()
    >>> for _ in range(3):
    ...     print(next(subsets))
    ...
    [[]]
    [[], [1]]
    [[], [1], [2], [1, 2]]
    """
    # YOUR-CODE-HERE
```

GENERATORS 1.5.3

```
def generate_subsets():
    # YOUR-CODE-HERE
```

Thought process:

- Uh…
- Okay, well it’s a generator function so we’re going to have to yield stuff
- Looking at the doctest, it seems as if we always want the positive integers to be in order. We’re just splitting them up
    - `[[]]`
    - `[[], [1]]`
    - `[[], [1], [2], [1, 2]]`
- What do you notice? Each successive yield is just *everything from before*, and also *everything from before* with the latest value of `n` tacked onto the end

GENERATORS 1.5.3

```
def generate_subsets():
    # YOUR-CODE-HERE
```

In other words,

- `n = 0:`
  `[[]]`
- `n = 1:`
  `[[], [1]]`, which is really just
  `[[] AND [] + [1]]`
- `n = 2:`
  `[[], [1], [2], [1, 2]]`, which is really just
  `[[], [1] AND [] + [2], [1] + [2]]`
- `n = 3:`
  `[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]`, or
  `[... AND [] + [3], [1] + [3], [2] + [3], [1, 2] + [3]]`

GENERATORS 1.5.3

```
def generate_subsets():
    n, subsets = 1, [[]]
    while True:
        yield subsets
        subsets += [s + [n] for s in subsets]
        n += 1
```

STREAMS

STREAMS

Streams are linked lists that are evaluated lazily:

- The rest won't be computed until we ask for it
- After we ask for it, the result will be **remembered**
- Rules [i.e. functions] will be used to compute the next element

Scheme stream interface:

- `car` gives us the first element of the stream
- `nil` is the empty stream
- `cons-stream`: like `cons`, except the rest isn't evaluated at first
- `cdr-stream`: like `cdr`, but tells the stream **to actually compute the rest** if it hasn't already. Note: don't use `cdr` with streams!

STREAMS

Non-required material, but you'll probably run into it so

Scheme promises (force/delay)

- How lazily evaluated expressions are actually implemented in Scheme
- Spec description
- **Promise**: a delayed expression
  - Can be "forced" (already evaluated and now cached)
  - or "not forced" (not evaluated yet)
- If you print a stream, you'll see this

```
scm> (cons-stream 1 2)
(1 . #[promise (not forced)])
```

STREAMS

What is the advantage of using a stream over a linked list?

STREAMS

What is the advantage of using a stream over a linked list?

Elements won't be evaluated unnecessarily if they are never used… meaning efficient space usage! Also, streams allow for the representation of infinite-length sequences.

On streams versus iterators:
Every time you call `next` on an iterator, it changes. The nice thing about streams is that they never change.

STREAMS E1

We attempt to define an infinite sequence of natural numbers.

```
(define (naturals n)
    (cons n (naturals (+ n 1))))
```

```
(define nat (naturals 0))
```

What happens?

STREAMS E1

We attempt to define an infinite sequence of natural numbers.

```
(define (naturals n)
    (cons n (naturals (+ n 1))))
```

```
(define nat (naturals 0))
```

```
Error: maximum recursion depth exceeded
```

STREAMS E1

That didn't work, so we turn to streams instead.

```
(define (naturals n)
    (cons-stream n (naturals (+ n 1))))
```

```
(define nat (naturals 0))

(car nat)

(car (cdr-stream nat))

(car (cdr-stream (cdr-stream nat)))
```

STREAMS E1

That didn't work, so we turn to streams instead.

```
(define (naturals n)
    (cons-stream n (naturals (+ n 1))))
```

```
(define nat (naturals 0)) → nat

(car nat) → 0

(car (cdr-stream nat)) → 1

(car (cdr-stream (cdr-stream nat))) → 2
```

STREAMS E1

```
(define (naturals n)
    (cons-stream n (naturals (+ n 1))))

(define nat (naturals 0))
```

```
(car nat)

(cdr nat)

(cdr-stream nat)

(cdr nat)
```

STREAMS E1

```
(define (naturals n)
    (cons-stream n (naturals (+ n 1))))

(define nat (naturals 0))
```

```
(car nat) → 0

(cdr nat) → #[promise (not forced)]

(cdr-stream nat) → (1 . #[promise (not forced)])

(cdr nat) → #[promise (forced)]
```

STREAMS 2.1.1 - WWSD?

```
scm> (define (has-even? s)
         (cond ((null? s) False)
               ((even? (car s)) True)
               (else (has-even? (cdr-stream s)))))
has-even?
scm> (define ones (cons-stream 1 ones))
ones
scm> (define twos (cons-stream 2 twos))
twos
scm> ones

scm> (cdr-stream ones)

scm> (has-even? ones)

scm> (has-even? twos)
```

STREAMS 2.1.1 - WWSD?

```
scm> (define (has-even? s)
         (cond ((null? s) False)
               ((even? (car s)) True)
               (else (has-even? (cdr-stream s)))))
has-even?
scm> (define ones (cons-stream 1 ones))
ones
scm> (define twos (cons-stream 2 twos))
twos
scm> ones
(1 . #[promise (not forced)])
scm> (cdr-stream ones)
(1 . #[promise (forced)])
scm> (has-even? ones)
Runs forever
scm> (has-even? twos)
True
```

STREAMS 2.1.2

Implement `map-stream`, which maps a function `f` to a stream `s`.

```
(define (map-stream f s)
    ; YOUR-CODE-HERE
)
```

```
scm> (define evens (map-stream (lambda (x) (* x 2)) nat))
evens
scm> (car (cdr-stream evens))
2
scm> (car (cdr-stream (cdr-stream evens)))
4
```

STREAMS 2.1.2

```
(define (map-stream f s)
    ; YOUR-CODE-HERE
)
```

Approach (what do we know?):

- We need to create a new stream
- We need to apply the function to every element
- We need to not think about the enormity of applying a function to every element in an infinite sequence, and instead consider the problem inductively

STREAMS 2.1.2

```
(define (map-stream f s)
    ; YOUR-CODE-HERE
)
```

Approach (what do we know?):

- We need to create a new stream
  - So we'll use `cons-stream`
- We need to apply the function to every element
  - So we'll do `(f (car s))`
- We need to not think about the enormity of applying a function to every element in an infinite sequence, and instead consider the problem inductively
  - So we'll recurse, creating the rest of the stream via `(map-stream f (cdr-stream s))`. Our base case will be an empty stream.

STREAMS 2.1.2

```
(define (map-stream f s)
    (if (null? s) nil
        (cons-stream (f (car s))
                     (map-stream f (cdr-stream s))))
)
```

STREAMS 2.1.3

Consider the following two implementations of `filter-stream`. One is correct and one is not. Which is the incorrect one, and why?

```
(define (filter-stream f s)
    (if (null? s) nil
        (let ((rest (filter-stream f (cdr-stream s))))
            (if (f (car s))
                (cons-stream (car s) rest)
                rest))))
```

```
(define (filter-stream f s)
    (cond ((null? s) nil)
          ((f (car s)) (cons-stream (car s) (filter-stream f (cdr-stream s))))
          (else (filter-stream f (cdr-stream s)))))
```

STREAMS 2.1.3

The first one is incorrect, because it recursively creates the rest of the stream either until it hits the end of s or until it hits the maximum recursion depth. We want lazy evaluation, so we shouldn't be constructing the entire stream at once!

```
(define (filter-stream f s)
    (if (null? s) nil
        (let ((rest (filter-stream f (cdr-stream s))))
            (if (f (car s))
                (cons-stream (car s) rest)
                rest))))

(define (filter-stream f s)
    (cond ((null? s) nil)
          ((f (car s)) (cons-stream (car s) (filter-stream f (cdr-stream s))))
          (else (filter-stream f (cdr-stream s)))))
```

STREAMS 2.1.4

Write a function, `range-stream`, that takes in two integers `start` and `end` and returns the same thing that `range(start, end)` would... but as a stream.

```
(define (range-stream start end)
    ; YOUR-CODE-HERE
)
```

```
scm> (define rs (range-stream 1 3))
rs
scm> (car rs)
1
scm> (car (cdr-stream rs))
2
scm> (cdr-stream (cdr-stream rs))
()
```

STREAMS 2.1.4

```
(define (range-stream start end)
    ; YOUR-CODE-HERE
)
```

Thought process:

- We have to create a stream where the first element is `start`
- The "rest" of the stream should be another stream whose first element is `start + 1`
- ^ Sounds like recursion to me. We can just make the rest of the stream be the return value of a `range-stream` call
- The base case, then, will be when `start` >= `end`. In this case, there's really no range to speak of – so we return an empty stream

STREAMS 2.1.4

```
(define (range-stream start end)
    ; YOUR-CODE-HERE
)
```

Thought process:

- We have to create a stream where the first element is `start`
  - `(cons-stream start ...)`
- The “rest” of the stream should be another stream whose first element is `start + 1`
  - `(cons-stream (+ start 1) ...)`
- ^ Sounds like recursion to me. We can just make the rest of the stream be the return value of a `range-stream` call
  - `(cons-stream start (range-stream (+ start 1) end))`
- The base case, then, will be when `start` >= `end`. In this case, there’s really no range to speak of – so we return an empty stream
  - `(if (>= start end) nil ...)`

STREAMS 2.1.4

```
(define (range-stream start end)
    (if (>= start end) nil
        (cons-stream start (range-stream (+ start 1) end)))
)
```

STREAMS 2.1.5

Write a function, `slice`, that returns a **list** containing the elements of `stream` from index `start` to index `end - 1`. If you run out of elements in `stream`, just cut the list short.

```
(define (slice stream start end)
    ; YOUR-CODE-HERE
)
```

STREAMS 2.1.5

```
(define (slice stream start end)
    ; YOUR-CODE-HERE
)
```

Thought process:

- We're returning a list! Back to `cons` and `list` and `append`… and no lazy evaluation. We need everything *now*
- Let's just recurse our way through `stream` decrementing `start` until it hits 0, and *then* start adding elements to our output list
- How will we know when to stop adding stuff? We want `end - start` elements overall… oh, we should also decrement `end` in that last step. Then, when `start` reaches 0, we'll know we want to add `end - 0` elements
- So we continue, decrementing `end` and adding elements to our list, until `end` is equal to 0. At this point, we'll be done

STREAMS 2.1.5

```
(define (slice stream start end)
    ; YOUR-CODE-HERE
)
```

Thought process, continued:

- We can use `cons` since it fits: we have an element and a “rest” list to use in the `(cons element rest)` formula
  - Specifically, we have `(cons (car stream) <recursive call>)`
- We should also check if `stream` is empty, since it’s not guaranteed that our indices are in-range

STREAMS 2.1.5

Bringing it all together, we have

```
(define (slice stream start end)
    (cond ((or (= end 0) (null? stream)) nil)
          ((> start 0) (slice (cdr-stream stream)
                              (- start 1)
                              (- end 1)))
          (else (cons (car stream)
                      (slice (cdr-stream stream)
                             0
                             (- end 1)))))
)
```

STREAMS 2.1.6

Let's combine infinite-length streams using `zip-with`, which accepts a function `f` and two streams (`xs` and `ys`) and returns a single stream containing every pair of corresponding elements combined with `f`.

If the elements in `xs` were `x1, x2, x3, …` and the elements in `ys` were `y1, y2, y3, …`, then `(zip-with f xs ys)` would return a stream with the elements `(f x1 y1), (f x2 y2), (f x3 y3), …`

```
(define (zip-with f xs ys)
    (if (or (null? xs) (null? ys)) nil
        (cons-stream (f (car xs) (car ys))
                     (zip-with f (cdr-stream xs) (cdr-stream ys)))))
)
```

STREAMS 2.1.6

As an example, we can create the even numbers by adding every natural number to Itself (s.t. the output sequence is 0 + 0, 1 + 1, 2 + 2, ...).

```
scm> (define evens (zip-with + (naturals 0) (naturals 0)))
evens
scm> (slice evens 0 10)
(0 2 4 6 8 10 12 14 16 18)
```

STREAMS 2.1.6

How would we define a stream containing the factorials of all numbers in order (starting at 0!) ? **Use `zip-with`!**

```
(define factorials
    ; YOUR-CODE-HERE
)
```

```
scm> (slice factorials 0 10)
(1 1 2 6 24 120 720 5040 40320 362880)
```

STREAMS 2.1.6

```
(define factorials
    ; YOUR-CODE-HERE
)
```

Thought process:

- `zip-with`? What does that do again? Oh yeah…
- Factorials? What are those again? Oh yeah…
  - 0! = 1
  - 1! = 1 * 0! = 1 * 1
  - 2! = 2 * 1! = 2 * 1 * 1
  - 3! = 3 * 2! = 3 * 2 * 1 * 1
- Interesting. It seems like we'd want to combine elements with * (multiplication)
- Well, based on the recurrences above, we'd want to multiply every natural number with the previous factorial. But we're *computing* the factorials! This is going to be trippy…

STREAMS 2.1.6

```
(define factorials
    (cons-stream 1 (zip-with * factorials (naturals 1)))
)
```

- `cons-stream` is kind of like a base case (it's the anchor; the first element of the output stream)
- Observing the sequence:
  - First element is 1 (obvious)
  - Second element is [^ that element] * 1 = 1 * 1 = 1
  - Third element is [^ that element] * 2 = 1 * 2 = 2
  - Fourth element is [^ that element] * 3 = 2 * 3 = 6
  - Fifth element is [^ that element] * 4 = 4 * 6
  - …
- I always thought this was neat af

STREAMS 2.1.6

Last one. How would we define a stream containing the Fibonacci numbers, starting with 0 and 1? **Use `zip-with`!**

```
(define fibs
    ; YOUR-CODE-HERE
)

scm> (slice fibs 0 10)
(0 1 1 2 3 5 8 13 21 34)
```

STREAMS 2.1.6

```
(define fibs
    ; YOUR-CODE-HERE
)
```

Let’s do this

- We’ve been computing Fibonacci numbers since day 0
  - F(0) = 0
  - F(1) = 1
  - F(2) = F(0) + F(1)
  - F(3) = F(1) + F(2)
  - …
- Looks like we’re gonna be combining with + (addition)
- Two “base cases”, 0 and 1, and then we’ll `(zip-with +)` a `fibs` and a staggered `fibs`

STREAMS 2.1.6

```
(define fibs
    (cons-stream 1 (cons-stream 2
            (zip-with + fibs (cdr-stream fibs))))
)
```

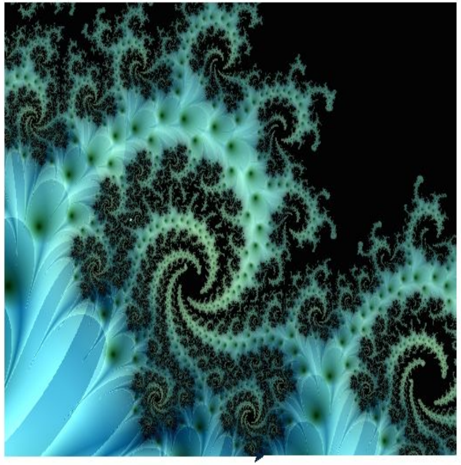
It's so beautiful...

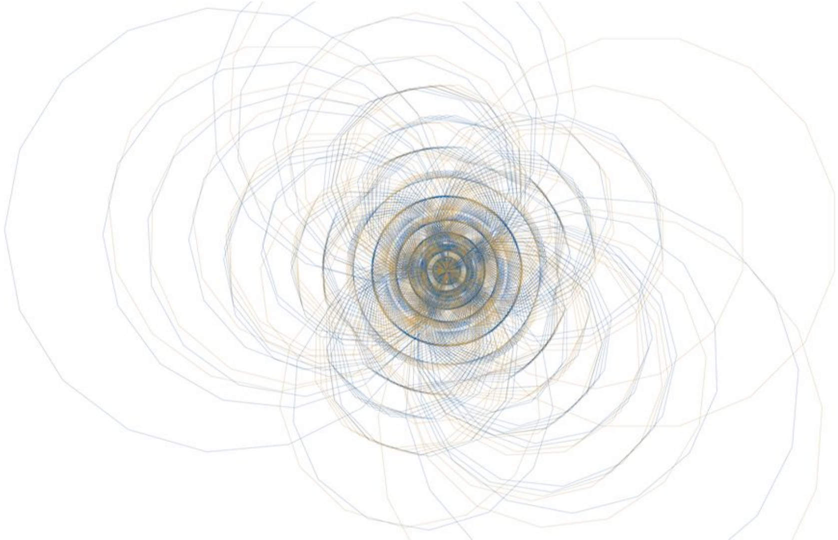
TURTLE

RECURSIVE ART CONTEST

It’s optional… but it’s also cool. Even if you don’t enter the contest, try playing around with the turtle graphics system for a bit!

Note: you can probably sell a lot of things as a recursive process





VECTOR GRAPHICS

Represent images using **geometric primitives** (points, lines, curves, shapes, polygons…) instead of **pixel values**. Based on **vectors** passing through control points, hence the name

As it happens, turtle graphics are vector-based!

DEMO: IMAGE VECTORIZATION

Something you can do with turtle graphics:

- Take a photograph (or grab any image you like)
- Convert the image to SVG format (https://www.vectorizer.io/)
- Implement turtle functions for all primitives in the VG format you’re using. This may only be `path` (a composite Bézier curve)!
- Convert the vector image into turtle code (using the primitives you just wrote) via a parser
- Let the turtle do its thing
- Profit

As I can show…

DISCUSSION ATTENDANCE

http://tiny.cc/5184

QUIZ 9