Lecture 9, 05 September 2022

Set comprehension

- Defining new sets from old
- $\{x^2 \mid x \in \mathbb{Z}, x \geq 0 \wedge (x \bmod 2) = 0\}$
  - $x \in \mathbb{Z}$, generating set
  - $x \geq 0 \wedge (x \bmod 2) = 0$, filtering condition
  - x^2, output transformation
- More generally $\{f(x) \mid x \in S, p(x)\}$
  - generating set S
  - filtering predicate $p()$
  - transformer function $f()$

Can do this manually for lists

- List of squares of even numbers from 0 to 19
- Initialize output list as `[]`
- Run through a loop and append elements to output list

In [1]:
```
evensqlist = []
for i in range(20):
  if i % 2 == 0:
    evensqlist.append(i*i)
print(evensqlist)
```

```
[0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
```

Operating on each element of a list

- `map(f,l)` applies a function `f` to each element of a list `l`
- `filter(p,l)` extracts elements `x` from `l` for which `p(x)` is `True

In [2]:
```
def even(x):
  return(x%2 == 0)

def odd(x):
  return(not(even(x)))

def square(x):
  return(x*x)

N = 20
l1 = list(range(N))
l2 = list(filter(odd,l1))  # Note that we can pass a function name as an argument
l3 = list(map(square,l1))

# Combine map and filter
l4 = list(map(square,filter(even,l1)))
```

In [3]:
```
l1
```

Out[3]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

In [4]:
```
l2
```

Out[4]:
```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

In [5]:
```
l3
```

Out[5]:
```
[0,
 1,
 4,
 9,
 16,
 25,
 36,
 49,
 64,
 81,
 100,
 121,
 144,
 169,
 196,
 225,
 256,
 289,
 324,
 361]
```

In [6]:
```
l4
```

Out[6]:
```
[0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
```

List comprehension

- `[ f(x) for x in ... if p(x) ]`

In [7]:
```
[ square(x) for x in range(20) if even(x) ]
```

Out[7]:
```
[0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
```

In [8]:
```
# A zero vector of length N
[ 0 for i in range(20)]   # The map function can be a constant function
```

Out[8]:
```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

- List comprehension can be nested
- A 2 dimensional list : A list of M lists of N zeros

In [9]:
```
M,N = 3,5
onedim = [ 0 for i in range(N)]    #  A list of N zeros
twodim = [ [0 for i in range(N)] for j in range(M)]
```

In [10]:
```
onedim, twodim
```

Out[10]:
```
([0, 0, 0, 0, 0], [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]])
```

All Pythagorean triples with value less than n

- (x, y, z) such that $x^2 + y^2 = z^2, x, y, z \leq n$

Using nested loops

- Run through all possible `(x,y,z)`
- To avoid duplicates like `(3,4,5)` and `(4,3,5)` enumerate `y` starting from `x`
- `z` must be at least `y` , enumerate `z` starting from `y`

In [11]:
```
N = 20
triples = []
for x in range(1,N+1):
    for y in range(x,N+1):
        for z in range(y,N+1):
            if x*x + y*y == z*z:
                triples.append((x,y,z))
```

In [12]:
```
triples
```

Out[12]:
```
[(3, 4, 5), (5, 12, 13), (6, 8, 10), (8, 15, 17), (9, 12, 15), (12, 16, 20)]
```

Pythagorean triples via list comprehension

- Multiple generators for `x`, `y` and `z`
- As before start generator for `y` at `x` and generator for `z` at `y`

In [13]:
```
N = 20
[ (x,y,z) for x in range(1,N+1) for y in range(x,N+1) for z in range(y,N+1) if x*x + y*y == z*z]
```

Out[13]:
```
[(3, 4, 5), (5, 12, 13), (6, 8, 10), (8, 15, 17), (9, 12, 15), (12, 16, 20)]
```

Uses of list comprehension

List comprehension notation is compact and useful in a number of contexts

- Pull out all dictionary values where the keys satisfy some property: e.g. all marks below 50
  - `[ d[k] for k in d.keys() if p(k) ]`
- Symmetrically, keys whose values satisfy some property: e.g. all roll numbers where marks are below 50
  - `[ k for k in d.keys() if p(d[k]) ]`
- Or, extract (key,value) pairs of interest
  - `[ (k,d[k]) for k in d.keys() if p(d[k]) ]`

Inductive definitions

- Define $f(n)$ in terms of n and $f(m)$ for $m < n$
- Need to define a base case explicitly, typically $f(0)$ or $f(1)$

Factorial

- $0! = 1$
- $n! = n \times (n-1)!$

Fibonacci numbers

- $fib(0) = 0$
- $fib(1) = 1$
- $fib(n) = fib(n-1) + fib(n-2)$

Recursive function calls

- A function can call itself
- Current execution is suspended until recursive call returns a value, like any other function call
- Recursive call will again call itself, so must ensure progress towards a base case for termination

In [14]:
```
def factorial(n):
  if n == 0:
    return(1)
  else:
    return(n*factorial(n-1))  # Recursive call
```

In [15]:
```
def fib(n):
  if n == 0:
    return(0)
  elif n == 1:
    return(1)
  else:
    return(fib(n-1)+fib(n-2))
```

Can also do induction on "structures"

- A list consists of the first element and the rest
- Base case is usually the empty list `[]`
- May occasionally also have a base case for a singleton list

In [16]:
```
def mylength(l):
  if l == []:
    return(0)
  else:
    return(1 + mylength(l[1:]))
```

In [17]:
```
mylength(list(range(900)))
```

Out[17]:
```
900
```

In [18]:
```
def mysum(l):
    if l == []:
        return(0)
    else:
        return(l[0] + mysum(l[1:]))
```

In [19]:
```
mysum(list(range(10)))
```

Out[19]:
```
45
```

In [20]:
```
mysum(['the','long','road'])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_216179/1486160608.py in <module>
----> 1 mysum(['the','long','road'])

/tmp/ipykernel_216179/1746613526.py in mysum(l)
      3         return(0)
      4     else:
----> 5         return(l[0] + mysum(l[1:]))

/tmp/ipykernel_216179/1746613526.py in mysum(l)
      3         return(0)
      4     else:
----> 5         return(l[0] + mysum(l[1:]))

/tmp/ipykernel_216179/1746613526.py in mysum(l)
      3         return(0)
      4     else:
----> 5         return(l[0] + mysum(l[1:]))

TypeError: can only concatenate str (not "int") to str
```

- Problem is `'the'+'long'+'road'+0`
- Could try to fix this by querying types from within the code, but we won't bother

Querying types

In [21]:
```
x = [3,3,4]
y = 5
type(x) == type([])  # Compare type() with a "known" type
```

Out[21]:
```
True
```

Can also compare `type(v)` with name of type without quotes

In [22]:
```
type(x) == list, type(y) == int
```

Out[22]:
```
(True, True)
```

Ascending and descending

- Check if a list is in ascending order, `l[0] < l[1] < ...`
- Similarly, descending, `l[0] > l[1] > ...`
- Ascending
  - Base case, `len(l)` is 0 or 1, nothing to check
  - Otherwise, check first pair `l[0] < l[1]`
  - Inductively check that that remaining list `l[1:]` is also ascending

In [23]:
```
def ascending(l):
  if len(l) <= 1:
    return(True)

  else:
    return(l[0] < l[1] and ascending(l[1:]))
    # l[0] < l[1] and l[1] < l[2] and l[2] < l[3] and ...

def descending(l):
  if len(l) <= 1:
    return(True)
  else:
    return(l[0] > l[1] and descending(l[1:]))
```

Zigzag

- Alternate between ascending and descending
- Two possibilities
  - up-down-up-down..., `[1,3,2,7,1,5]`
  - down-up-down-up..., `[8,2,18,-5,7,2,8]`

Up-down

- If `len(l)` is 0 or 1, nothing to do
- Up-down unit repeats after two elements
- 2 element list, check up
- 3 element list, check up-down and recursively check that `l[2:]` is also up-down

Down-up is symmetric

Combine to get zigzag

In [24]:
```
def updown(l):
  if len(l) <= 1:
    return(True)
  elif len(l) == 2:
    return(l[0] < l[1])
  else:
    return(l[0] < l[1] and l[1] > l[2] and updown(l[2:]))

def downup(l):
  if len(l) <= 1:
    return(True)
  elif len(l) == 2:
    return(l[0] > l[1])
  else:
    return(l[0] > l[1] and l[1] < l[2] and downup(l[2:]))

def zigzag(l):
  return(updown(l) or downup(l))
```

In [25]:
```
l1 = [1,2,1,3,1,4,1]
updown(l1), downup(l1), zigzag(l1)
```

Out[25]:
```
(True, False, True)
```

In [26]:
```
l2 = [2,1,3,1,4,1]
updown(l2), downup(l2), zigzag(l2)
```

Out[26]:
```
(False, True, True)
```

Mutual recursion

- Can define `updown` and `downup` in terms of each other
- **Mutual recursion**

In [27]:
```
def zigzag(l):
    return(updown(l) or downup(l))

def updown(l):
    if len(l) < 2:
        return(True)
    else:
        return(l[0] < l[1] and downup(l[1:]))

def downup(l):
    if len(l) < 2:
        return(True)
    else:
        return(l[0] > l[1] and updown(l[1:]))
```

In [28]:
```
zigzag([0,1,0,1,0])
```

Out[28]:
```
True
```