# DM502
# Programming A

Peter Schneider-Kamp

petersk@imada.sdu.dk

http://imada.sdu.dk/~petersk/DM502/

# PROJECT PART I

# Organizational Details

- 2 possible projects, each consisting of 2 parts
- for 1st part, you have to pick ONE
- for 2nd part, you can stay or you may switch

- projects must be done individually, so no co-operation
- you may talk about the problem and ideas how to solve them

- deliverables:
  - written 4 page report as specified in project description
  - handed in BOTH electronically and as paper
  - deadline: September 30, 12:00

- ENOUGH - now for the FUN part …

# Fractals and the Beauty of Nature

- geometric objects similar to themselves at different scales

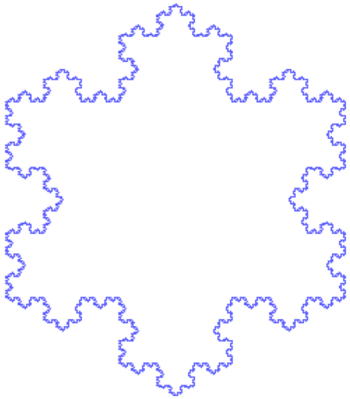
- many structures in nature are fractals:
  - snowflakes
  - lightning
  - ferns

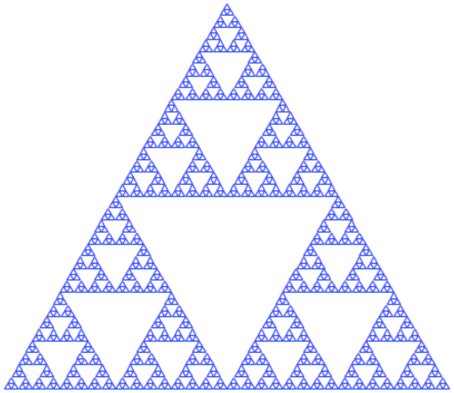
- **Goal:** generate fractals using Swampy

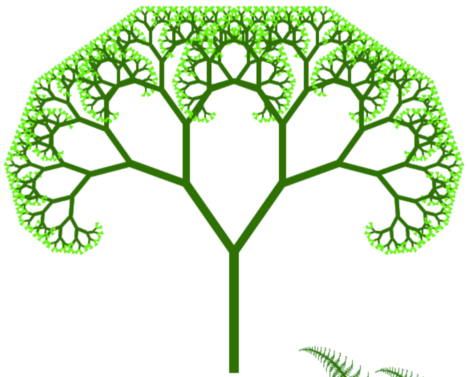
- **Challenges:** Recursion, Tuning, Library Use

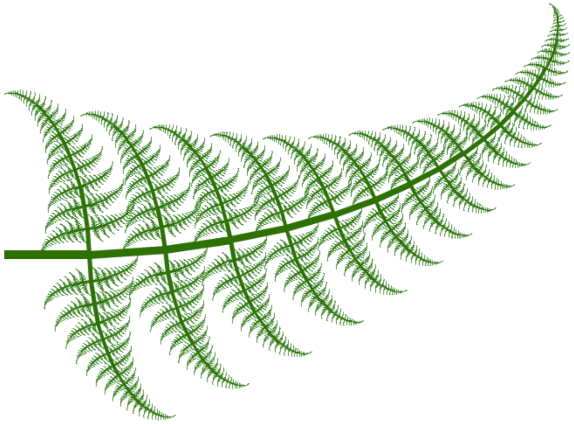
# Fractals and the Beauty of Nature

- Task 0: Preparation
  - understand implementation of Koch snowflake
- Task 1: Sierpinski Triangle
  - draw fractal triangle of fixed depth
- Task 2: Binary Tree
  - draw binary trees of fixed depth
- Task 3 (optional): Fern Time
  - draw beautiful fern leaves with fixed detail





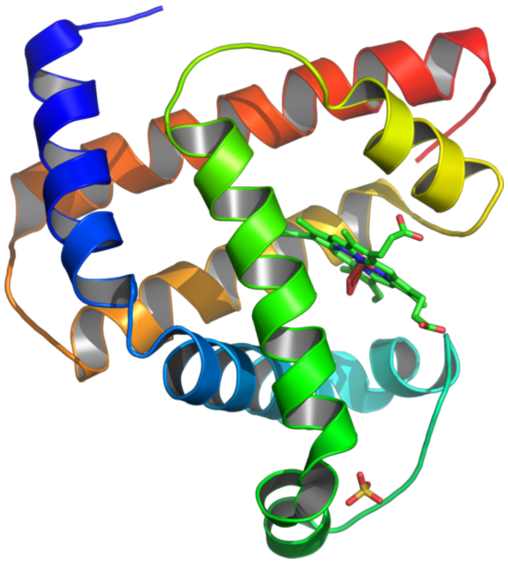




# From DNA to Proteins

- proteins encoded by DNA base sequence using A, C, G, and T

- Background:
  - proteins are sequences of amino acids
  - amino acids encoded using three bases
  - chromosomes given as base sequences



- **Goal:** assemble and analyze sequences from files

- **Challenges:** File Handling, String and List Methods, Iteration

# From DNA to Proteins

- Task 0: Preparation
  - download human DNA sequence and take a look at it
- Task 1: Assembling the Sequence
  - clean up the sequence and assemble it into one string
- Task 2: Finding Starting Points
  - find positions in string where ATG closely follows TATAAA
- Task 3: Finding End Points
  - find one of the potential end markers (TAG, TAA, TGA)
- Task 4 (optional): Potential Proteins without TATA Boxes
  - analysis of overlaps in encoded proteins

# LIST PROCESSING

# Lists as Sequences

- lists are sequences of values
- lists can be constructed using "[" and "]"
- Example:
```
[42, 23]
["Hello", "World", "!"]
["strings and", int, "mix", 2]
[]
```
- lists can be nested, i.e., a list can contain other lists
- Example:
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```
- lists are normal values, i.e., they can be printed, assigned etc.
- Example:
```
x = [1, 2, 3]
print x, [x, x], [[x, x], x]
```

# Mutable Lists

- lists can be accessed using indices
- lists are mutable, i.e., they can be changed destructively
- Example:

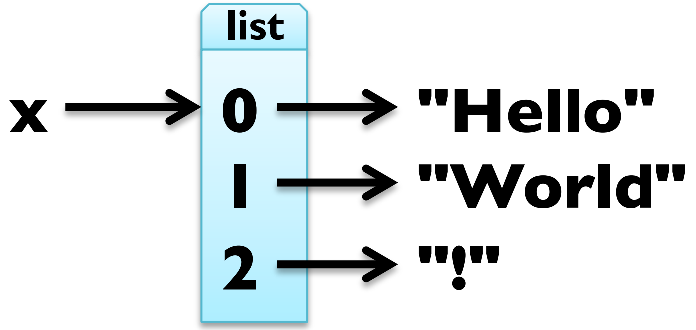
```
x = [1, 2, 3]
print x[1]
x[1] = 4
print x, x[1]
```

- len(object) and negative values work like for strings
- Example:

```
x[2] == x[-1]
x[1] == x[len(x)-2]
```

# Stack Diagrams with Lists

- lists can be viewed as mappings from indices to elements
- Example 1: x = ["Hello", "World", "!"]




- Example 2: x = [[23, 42, -3.0], "Bye!"]

# Traversing Lists

- for loop consecutively assigns variable to elements of list
- Example: print squares of numbers from 1 to 10

```
for x in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    print x**2
```

- arithmetic sequences can be generated using range function:
  - range([start,] stop[, step])
- Example:

```
range(4) == [0, 1, 2, 3]
range(1, 11) == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
range(9, 1, -2) == [9, 7, 5, 3]
range(1,10, 2) == [1, 3, 5, 7, 9]
```

# Traversing Lists

- for loop consecutively assigns variable to elements of list
- general form

```
for element in my_list:
    print element
```

- iteration through list with indices:

```
for index in range(len(my_list)):
    element = my_list[index]
    print element
```

- Example: in-situ update of list

```
x = [8388608, 43980465111104, 0.125]
for i in range(len(x)):
    x[i] = math.log(x[i], 2)
```

# List Operations

- like for strings, "+" concatenates two lists
- Example:

```
[1, 2, 3] + [4, 5, 6] == range(1, 7)
[[23, 42] + [-3.0]] + ["Bye!"] == [[23, 42, -3.0], "Bye!"]
```

- like for strings, "* n" with integer n produces n copies
- Example:

```
len(["I", "love", "penguins!"] * 100) == 300
(range(1, 3) + range(3, 1, -1)) * 2 == [1, 2, 3, 2, 1, 2, 3, 2]
```

# List Slices

- slices work just like for strings
- Example:
```
x = ["Hello", 2, "u", 2, "!"]
x[2:4] == ["u", 2]
x[2:] == x[-3:len(x)]
y = x[:]          # make a copy (lists are mutable!)
```
- BUT: we can also assign to slices!
- Example:
```
x[1:4] = ["to", "you", "too"]
x == ["Hello", "to", "you", "too", "!"]
x[1:3] = ["to me"]
x == ["Hello", "to me", "too", "!"]
x[2:3] = []
x == ["Hello", "to me", "!"]
```

# List Methods

- appending elements to the end of the list (destructive)
- Example: x = [5, 3, 1]

  y = [2, 4, 6]

  for e in y: x.append(e)
- Note: x += [e] would create new list in each step!
- also available as method: x.extend(y)

- sorting elements in ascending order (destructive)
- Example: x.sort()

  x == range(1, 7)

- careful with destructive updates: x = x.sort()

# Higher-Order Functions (map)

- Example 1: new list with squares of all elements of a list

```
def square_all(x):
    res = []
    for e in x:    res.append(e**2)
    return res
```

- Example 2: new list with all elements increased by one

```
def increment_all(x):
    res = []
    for e in x:    res.append(e+1)
    return res
```

# Higher-Order Functions (map)

- these *map* operations have an identical structure:

```
res = []
for e in x:   res.append(e**2)
return res
```

```
res = []
for e in x:   res.append(e+1)
return res
```

- Python has generic function map(function, sequence)
- Implementation idea:

```
def map(function, sequence):
    res = []
    for e in sequence:
        res.append(function(e))
    return res
```

# Higher-Order Functions (map)

- these *map* operations have an identical structure:

```
res = []                                   res = []
for e in x:   res.append(e**2)             for e in x:   res.append(e+1)
return res                                 return res
```

- Python has generic function map(function, sequence)
- Example:

```
def square(x):          return x**2
def increment(x):       return x+1
def square_all(x):
    return map(square, x)
def increment_all(x):
    return map(increment, x)
```

# Higher-Order Functions (filter)

- Example 1: new list with elements greater than 42

```
def filter_greater42(x):
    res = []
    for e in x:
        if e > 42:    res.append(e)
    return res
```

- Example 2: new list with elements whose length is smaller 3

```
def filter_len_smaller3(x):
    res = []
    for e in x:
        if len(e) < 3:   res.append(e)
    return res
```

# Higher-Order Functions (filter)

- these *filter* operations have an identical structure:

```
res = []
for e in x:
    if e > 42:   res.append(e)
return res
```

```
res = []
for e in x:
    if len(e) < 3:   res.append(e)
return res
```

- Python has generic function filter(function, iterable)
- Implementation idea:

```
def filter(function, iterable):
    res = []
    for e in iterable:
        if function(e):     res.append(e)
    return res
```

# Higher-Order Functions (filter)

- these *filter* operations have an identical structure:

```
res = []
for e in x:
    if e > 42:   res.append(e)
return res
```

```
res = []
for e in x:
    if len(e) < 3:   res.append(e)
return res
```

- Python has generic function filter(function, iterable)
- Example:

```
def greater42(x):              return x > 42
def len_smaller3(x):           return len(x) < 3
def filter_greater42(x):       return filter(greater42,  x)
def filter_len_smaller3(x):    return filter(len_smaller3, x)
```

# Higher-Order Functions (reduce)

- Example 1: computing factorial using range

```
def mul_all(x):
    prod = 1
    for e in x:   prod *= e          # prod = prod * e
    return prod
def factorial(n):
    return mul_all(range(1,n+1))
```

- Example 2: summing all elements in a list

```
def add_all(x):
    sum = 0
    for e in x:   sum += e          # sum = sum + e
    return sum
```

# Higher-Order Functions (reduce)

- these *reduce* operations have an identical structure:

```
prod = 1                          sum = 0
for e in x:   prod *= e           for e in x:   sum += e
return prod                       return sum
```

- Python has generic function reduce(function, sequence, initial)
- Implementation idea:

```
def reduce(function, sequence, initial):
    result = initial
    for e in sequence:
        result = function(result, e)
    return result
```

# Higher-Order Functions (reduce)

- these *reduce* operations have an identical structure:

```
prod = 1                         sum = 0
for e in x:   prod *= e          for e in x:      sum += e
return prod                      return sum
```

- Python has generic function reduce(function, sequence, initial)
- Example:

```
def add(x,y):   return x+y
def mul(x,y):   return x*y
def add_all(x):
    return reduce(add, x, 0)
def mul_all(x):
    return reduce(mul, x, 1)
```

# Deleting Elements

- there are three different ways to delete elements from list

- if you know index and want the element, use pop(index)
- Example: my_list = [23, 42, -3.0, 4711]
  my_list.pop(1) == 42
  my_list == [23, -3.0, 4711]
- if you do not know index, but the element, use remove(value)
- Example: my_list.remove(-3.0)
  my_list == [23, 4711]
- if you know the index, you can use the del statement
- Example: del my_list[0]
  my_list == [4711]

# Deleting Elements

- there are three different ways to delete elements from list

- as we have seen, you can also use slices to delete elements
- Example:

```
my_list = [23, 42, -3.0, 4711]
my_list[2:] = []
my_list == [23, 42]
```

- alternatively, you can use del together with slices
- Example:

```
my_list = my_list * 3
del my_list[:3]
my_list == [42, 23, 42]
```

# Lists vs Strings

- string = sequence of letters
- list = sequence of values

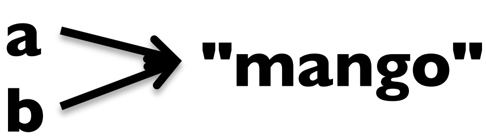
- convert a string into a list using the built-in list() function
- Example: list("Hej hop") == ["H", "e", "j", " ", "h", "o", "p"]

- split up a string into a list using the split(sep) method
- Example: "Slartibartfast".split("a") == ["Sl", "rtib", "rtf", "st"]

- reverse operation is the join(sequence) method
- Example: " and ".join(["A", "B", "C"]) == "A and B and C"
  "".join(["H", "e", "j", " ", "h", "o", "p"]) = "Hej Hop"

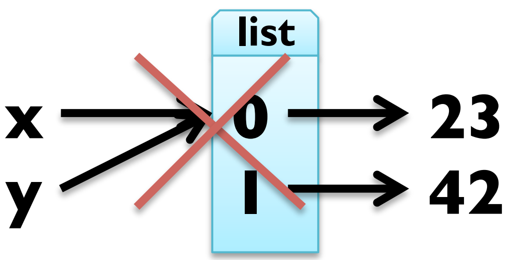
# Objects and Values

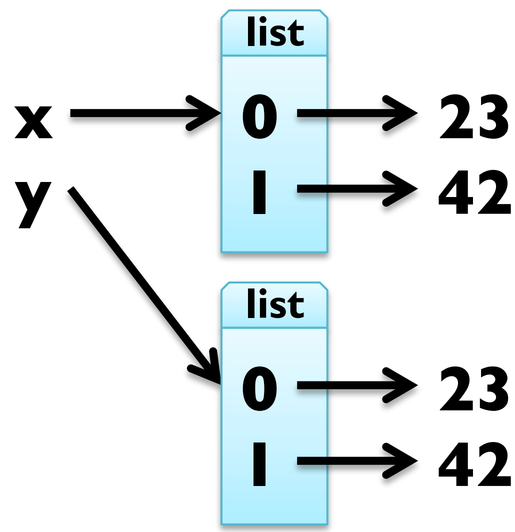
- two possible stack diagrams for a = "mango"; b = "mango"





- we can check identity of objects using the is operator
- Example: a is b == True
- two possible stack diagrams for x = [23, 42]; y = [23, 42]

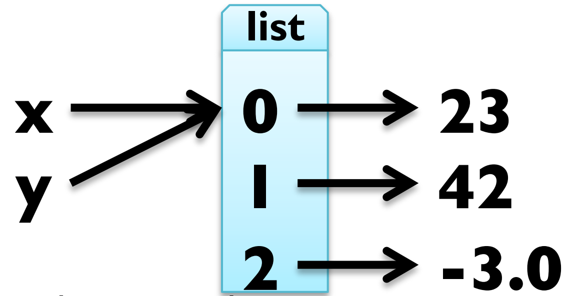



- Example: x is y == False

# Aliasing

- when assigning y = x, both variables refer to same object
- Example: x = [23, 42, -3.0]
  y = x
  x is y == True




- here, there are two *references* to one (*aliased*) object

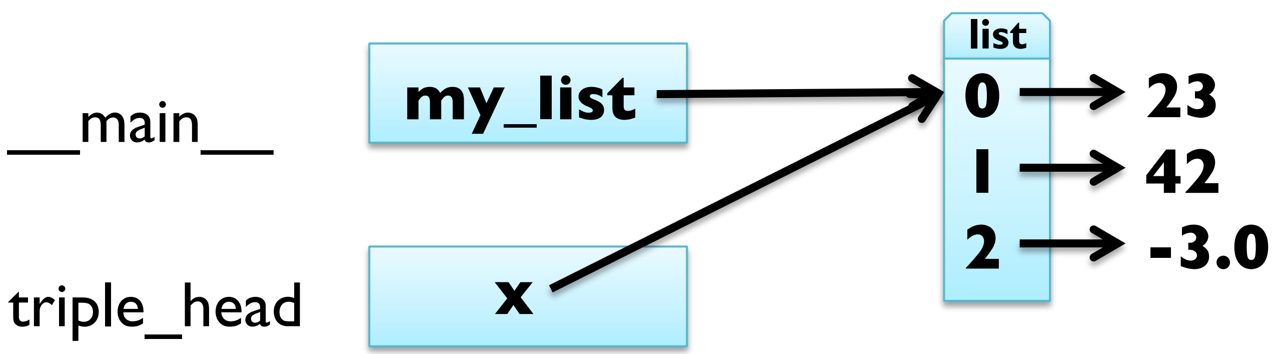
- fine for immutable objects (like strings)
- problematic for mutable objects (like lists)
- Example: y[2] = 4711
  x == [23, 42, 4711]

- HINT: when unsure, always copy list using y = x[:]

# List Arguments

- lists passed as arguments to functions can be changed
- Example: tripling the first element

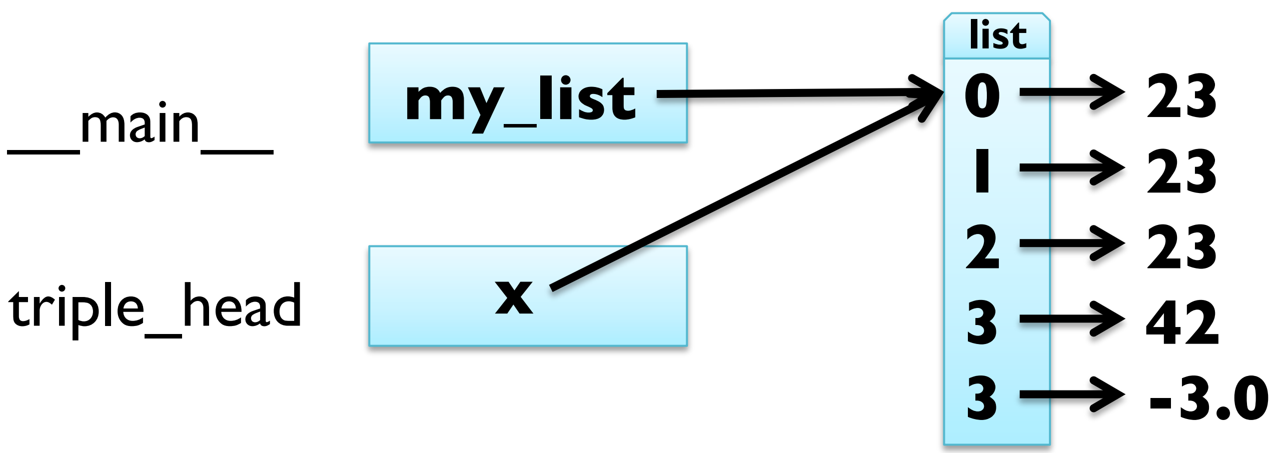
```
def triple_head(x):
    x[:1] = [x[0]]*3
my_list = [23, 42, -3.0]
triple_head(x)
```



# List Arguments

- lists passed as arguments to functions can be changed
- Example: tripling the first element

```
def triple_head(x):
    x[:1] = [x[0]]*3
my_list = [23, 42, -3.0]
triple_head(x)
my_list == [23, 23, 23, 42, -3.0]
```



# List Arguments

- lists passed as arguments to functions can be changed
- some operations change object
  - assignment using indices
  - append(object) method
  - extend(iterable) method
  - sort() method
  - del statement
- some operations return a new object
  - access using slices
  - strip() method
  - “+” on strings and lists
  - “* n” on strings and lists

# Debugging Lists

- working with mutable objects like lists requires attention!

1. many list methods return None and modify destructively
   - word = word.strip() makes sense
   - t = t.sort() does NOT!
2. there are many ways to do something – stick with one!
   - t.append(x) or t = t + [x]
   - use either pop, remove, del or slice assignment for deletion
3. make copies when you are unsure!
   - Example:

```
…
sorted_list = my_list[:]
sorted_list.sort()
…
```

# DICTIONARIES

# Generalized Mappings

- list = mapping from integer indices to values
- dictionary = mapping from (almost) any type to values
- indices are called *keys* and pairs of keys and values *items*

- empty dictionaries created using curly braces "{}"
- Example: `en2da = {}`

- keys are assigned to values using same syntax as for sequences
- Example:
```
en2da["queen"] = "dronning"
print en2da
```

- curly braces "{" and "}" can be used to create dictionary
- Example: `en2da = {"queen" : "dronning", "king" : "konge"}`

# Dictionary Operations

- printing order can be different: print en2da
- access using indices: en2da["king"] == "konge"
- KeyError when key not mapped: print en2da["prince"]
- length is number of items: len(en2da) == 2
- in operator tests if key mapped: "king" in en2da == True
  "prince" in en2da == False
- keys() metod gives list of keys:
  en2da.keys() == ["king", "queen"]
- values() method gives list of values:
  en2da.values() == ["konge", "dronning"]
- useful e.g. for test if value is used:
  "prins" in en2da.values() == False