# FIT1045: Algorithms and Programming Fundamentals in Python

## Lecture 5

## Collections

https://www.invaluable.com/blog/collecting-psychology/

# Announcements

- Assignment release tomorrow
- Test 1 next week

# Recap: while loops

```
def func(n):
    i = 1
    res = 0
    while i <= n:
        res = res + i
        i = i + 1
    return res
```

```
>>> func(10)
?
```

https://flux.qa

Clayton: AXXULH
Malaysia: LWERDE

# Goal this week: use Python to track macro-nutrients

**Input:**

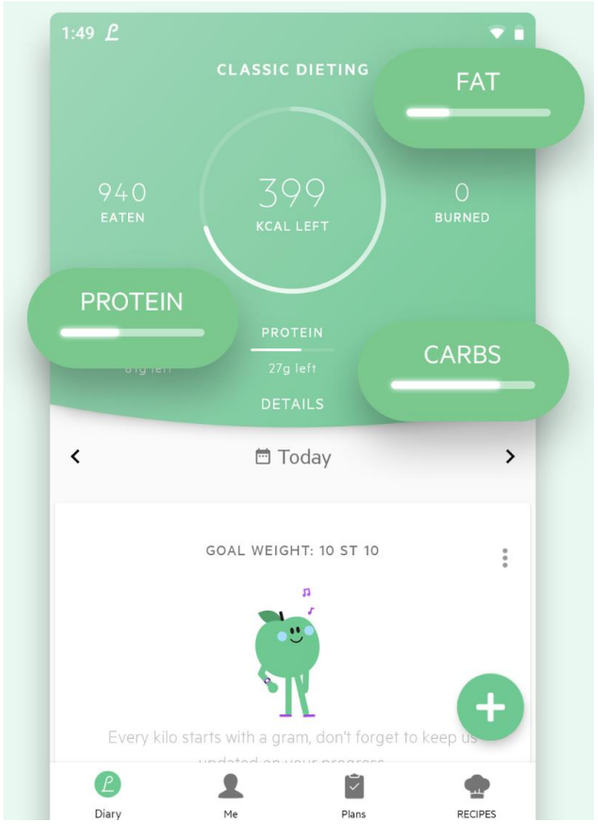
| id | day | food | quantity |
|---|---|---|---|
| 1 | 1 | beef | 300 |
| 2 | 1 | potato | 300 |
| 3 | 1 | broccoli | 200 |
| 4 | 1 | apple | 100 |
| 5 | 2 | potato | 250 |
| 6 | 2 | apple | 100 |
| 7 | 2 | tofu | 120 |
| 8 | 2 | tomato | 200 |
| 9 | 3 | rice | 220 |
| 10 | 3 | carrot | 120 |
| 11 | 3 | eggplant | 150 |
| 12 | 3 | coconut cream | 160 |
| 13 | 3 | apple | 110 |

food diary

| food | energy | water | protein | carbs | sugars | fat | fibres |
|---|---|---|---|---|---|---|---|
| apple | 229 | 84.3 | 0.4 | 12 | 11.8 | 0 | 2.3 |
| orange | 186 | 84.3 | 1 | 9.5 | 8.3 | 0.2 | 2.1 |
| broccoli | 124 | 89.6 | 3.2 | 2 | 2 | 0.1 | 4.1 |
| beef | 613 | 70 | 22.8 | 0.2 | 0 | 6 | 0 |
| lamb | 1057 | 60.2 | 18.6 | 0 | 0 | 20.2 | 0 |
| bread | 1446 | 37.6 | 8.4 | 43.5 | 1.5 | 2.6 | 6.9 |
| potato | 346 | 77.4 | 2 | 17 | 0 | 0.1 | 2.5 |
| tofu | 510 | 74 | 12 | 1.5 | 0.5 | 6.5 | 5 |
| tomato | 81 | 93.3 | 1 | 2.9 | 0.9 | 0.2 | 1 |
| eggplant | 107 | 91.6 | 1.2 | 3.5 | 1.5 | 0.2 | 2.5 |
| carrot | 116 | 90.6 | 0.8 | 4.7 | 4.4 | 0 | 2.9 |
| coco. cream | 872 | 73 | 1.5 | 3 | 0 | 21.5 | 0 |
| rice | 403 | 75.3 | 2.5 | 20 | 0 | 0.4 | 0.8 |

database of nutrition values

**Output:**




https://play.google.com/store/apps/details?id=com.sillens.shapeupclub

nutritional intake per day

# Objectives

Being able to represent an manipulate *complex* inputs

- sequences and other collection objects
- for-loops to investigate collections
- ranges for solving problems with for-loops

Learning outcomes

- 1 (translate between problem descriptions and program design with appropriate input/output representations)
- 2 (choose and implement appropriate problem solving strategies in Python )

**Concrete goal (this week)**: nutrition app

# Where am I?

1. Collections
2. For-loops
3. Ranges

# Recap: sequence objects

```
>>> items = ['milk', 'eggs', 'bread', 'jam', 'bread']
>>> items
['milk', 'eggs', 'bread', 'jam', 'bread']
>>> len(items)
5
>>> 'bread' in items
True
>>> len('eggs')
4
>>> 'g' in 'eggs'
True
```

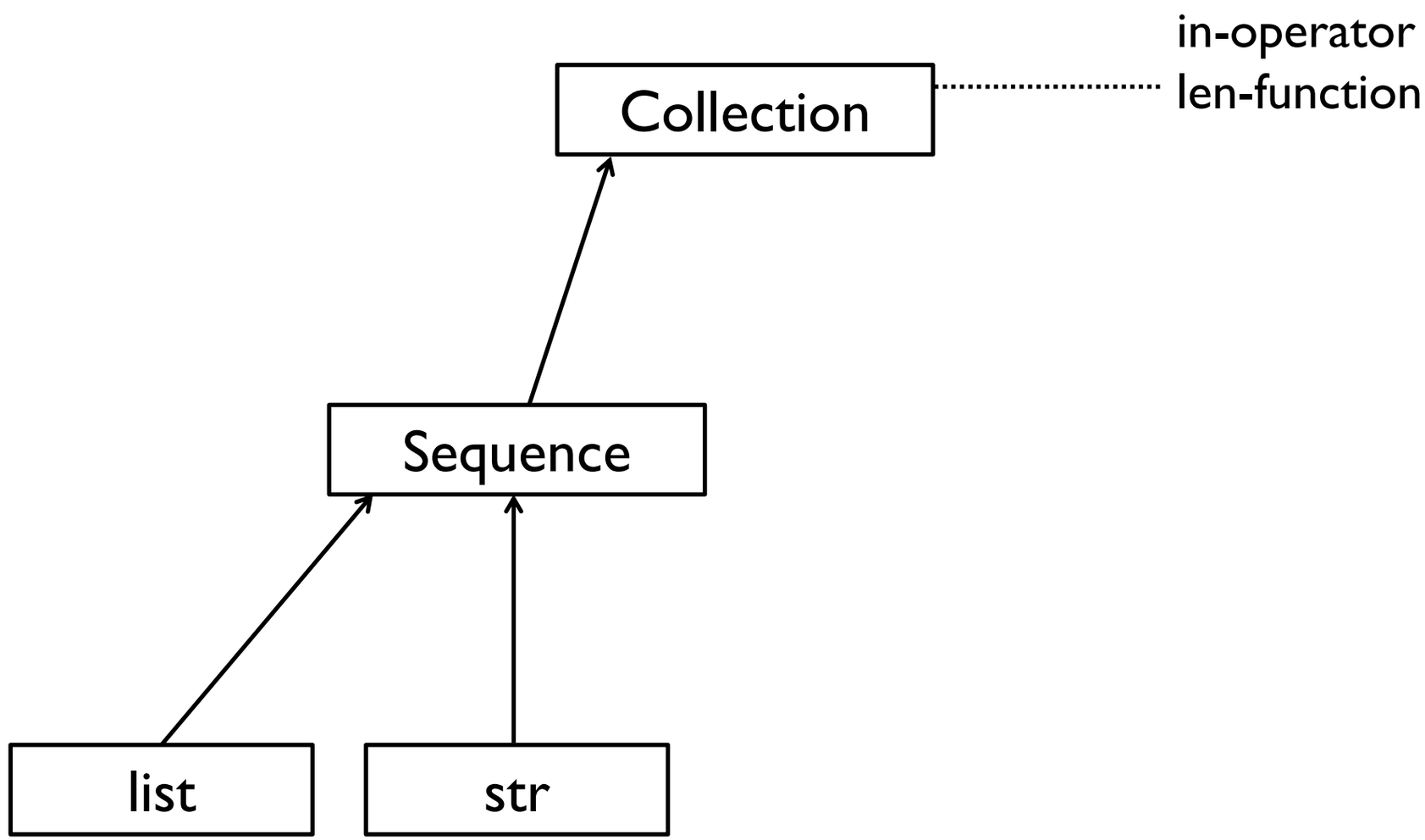
String and list objects are sequences

- Sequences are *ordered* "collections" of other objects
- Can contain identical objects multiple times

Sequences are collections

- The *in*-operator tests whether object is contained
- The *len*-function returns number of contained objects (counting each repetition of object)

# Collections type hierarchy



# Access to sequence elements via *indexing operator*

```
>>> items = ['milk', 'eggs', 'bread', 'jam', 'bread']
>>> items[1]
'eggs'
>>> items[0]
'milk'
>>> items[len(items) - 1]
'bread'
>>> items[len(items)]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

- Python indexing is zero-based, i.e., first element at index 0
- So the last element is at len(x)-1
- Index greater than that is invalid: IndexError

| **value** | 'milk' | 'eggs' | 'bread' | 'jam' | 'bread' |
|---|---|---|---|---|---|
| **index** | 0 | 1 | 2 | 3 | 4 |

# Negative indices are allowed

```
>>> items = ['milk', 'eggs', 'bread', 'jam', 'bread']
>>> items[-1]
'bread'
>>> items[-2]
'jam'
>>> items[-len(items)]
'milk'
>>> items[-(len(items)+1)]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

- Negative indices access sequence from the back
- Smallest valid index is -len(x) addressing the first element
- Index less than that again invalid: IndexError

| value | 'milk' | 'eggs' | 'bread' | 'jam' | 'bread' |
|---|---|---|---|---|---|
| index | 0 | 1 | 2 | 3 | 4 |
| negative index | -5 | -4 | -3 | -2 | -1 |

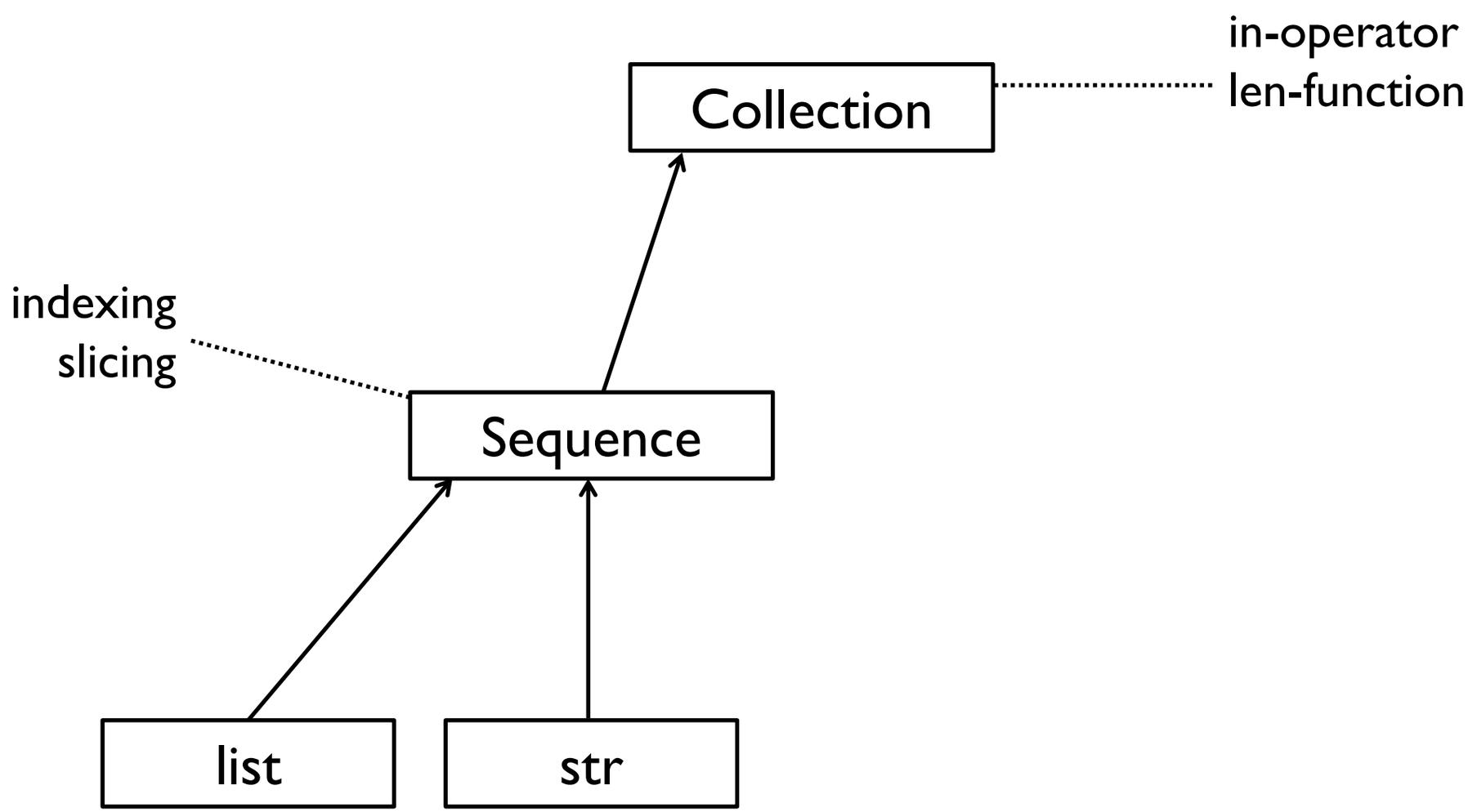
# Sub-sequences are available via *slicing* operator

```
>>> items = ['milk', 'eggs', 'bread', 'jam', 'bread']
>>> items[1:3]
['eggs', 'bread']
>>> items[1:len(items)]
['eggs', 'bread', 'jam', 'bread']
>>> items[1:]
['eggs', 'bread', 'jam', 'bread']
>>> items[0:3]
['milk', 'eggs', 'bread']
>>> items[:3]
['milk', 'eggs', 'bread']
>>>
```

- Slice specified by start index (inclusive) and stop index (exclusive)
- Either index can be omitted (defaults to 0 and len(x))

| **value** | 'milk' | 'eggs' | 'bread' | 'jam' | 'bread' |
|---|---|---|---|---|---|
| **index** | 0 | 1 | 2 | 3 | 4 |
| **negative index** | -5 | -4 | -3 | -2 | -1 |

# Collections type hierarchy

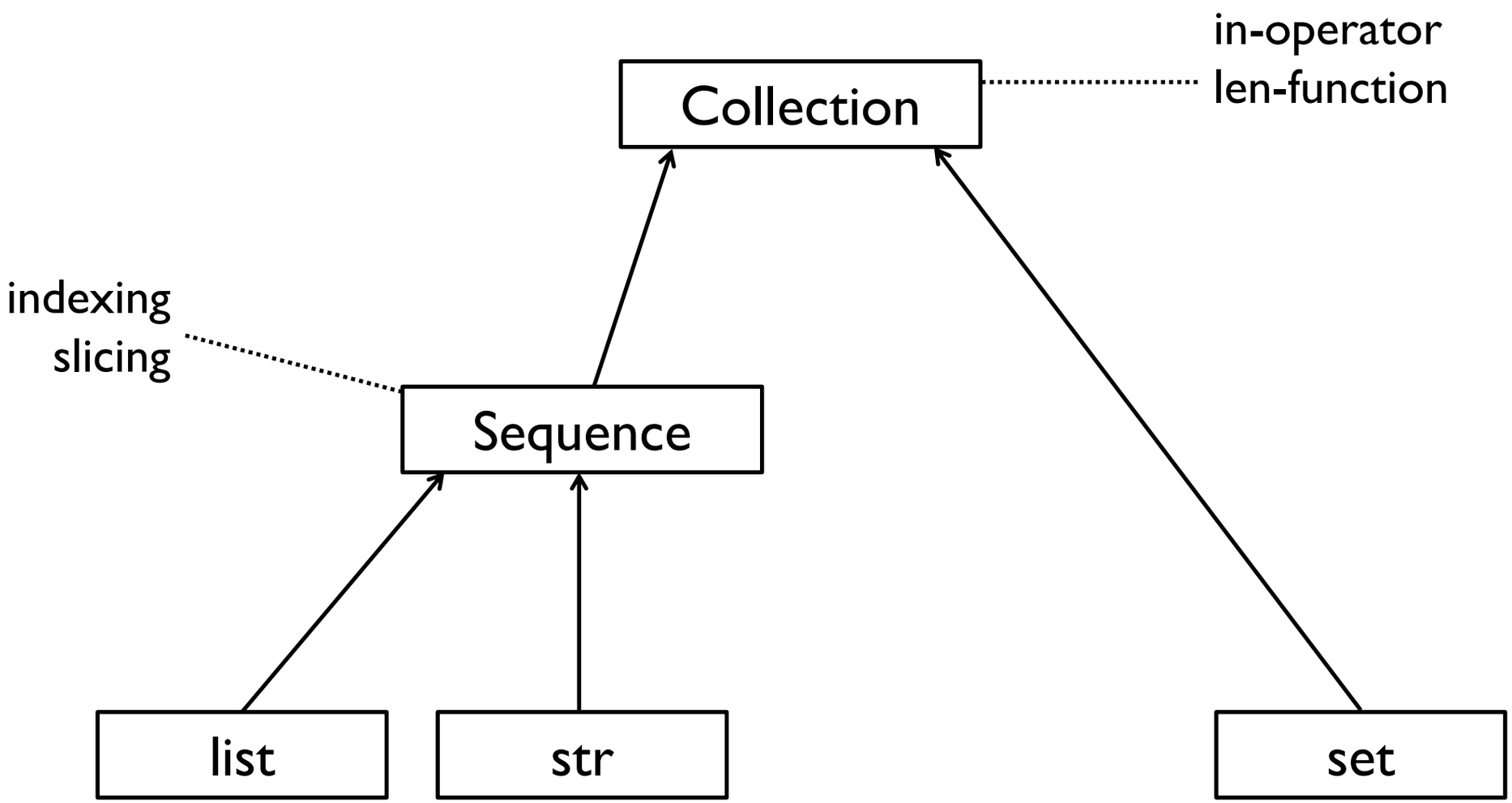


# *Sets*: another collection type

```
>>> items = {'milk', 'eggs', 'bread', 'jam', 'bread'}
>>> type(items)
<class 'set'>
>>> 'bread' in items
True
>>> len(items)
4
>>> items
{'bread', 'milk', 'jam', 'eggs'}
>>> items[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

- Sets are *unordered* containers of other objects
- Can contain equal objects only once
- Cannot be indexed (or sliced)

# Collections type hierarchy

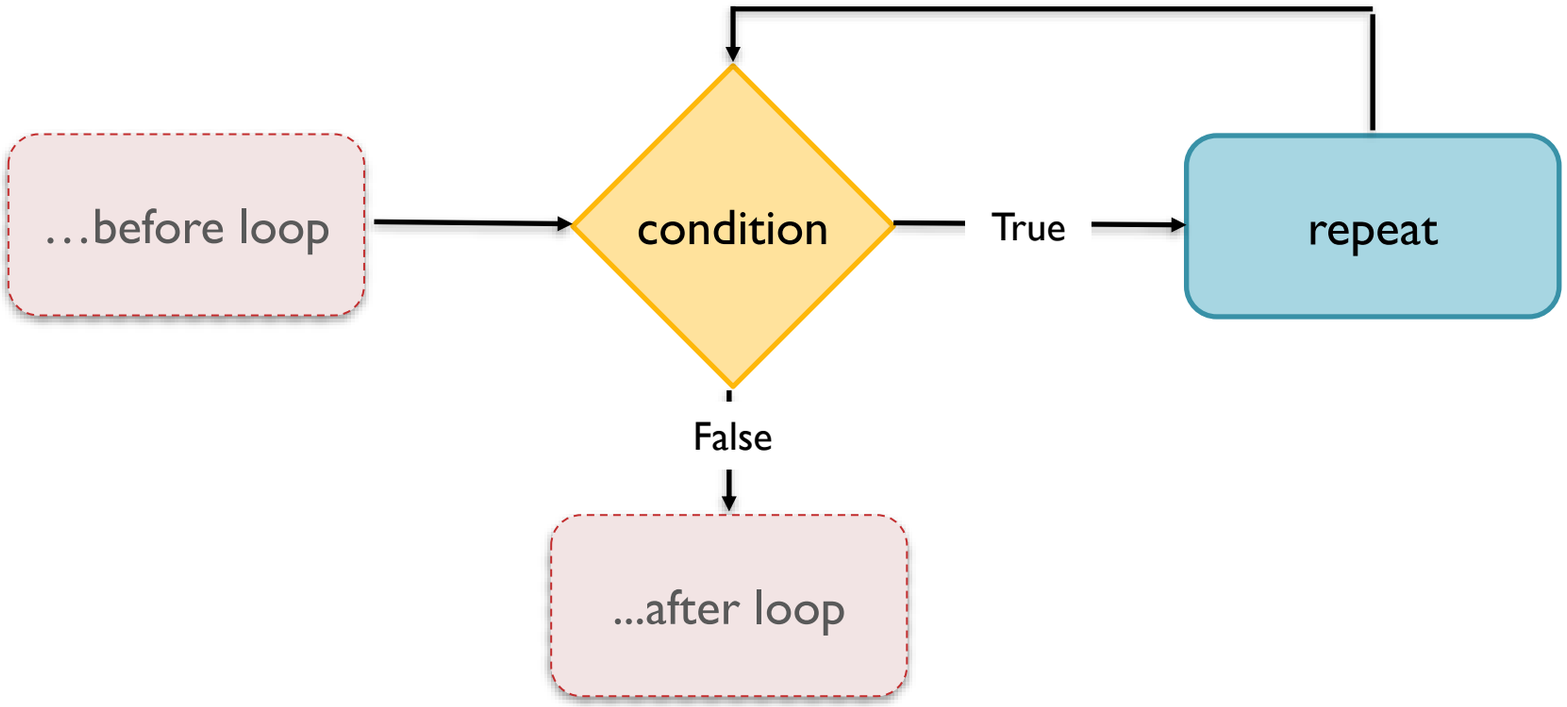


# Where am I?

1. Collections
2. For-loops
3. Ranges

# Problem: count occurrences of object in sequence

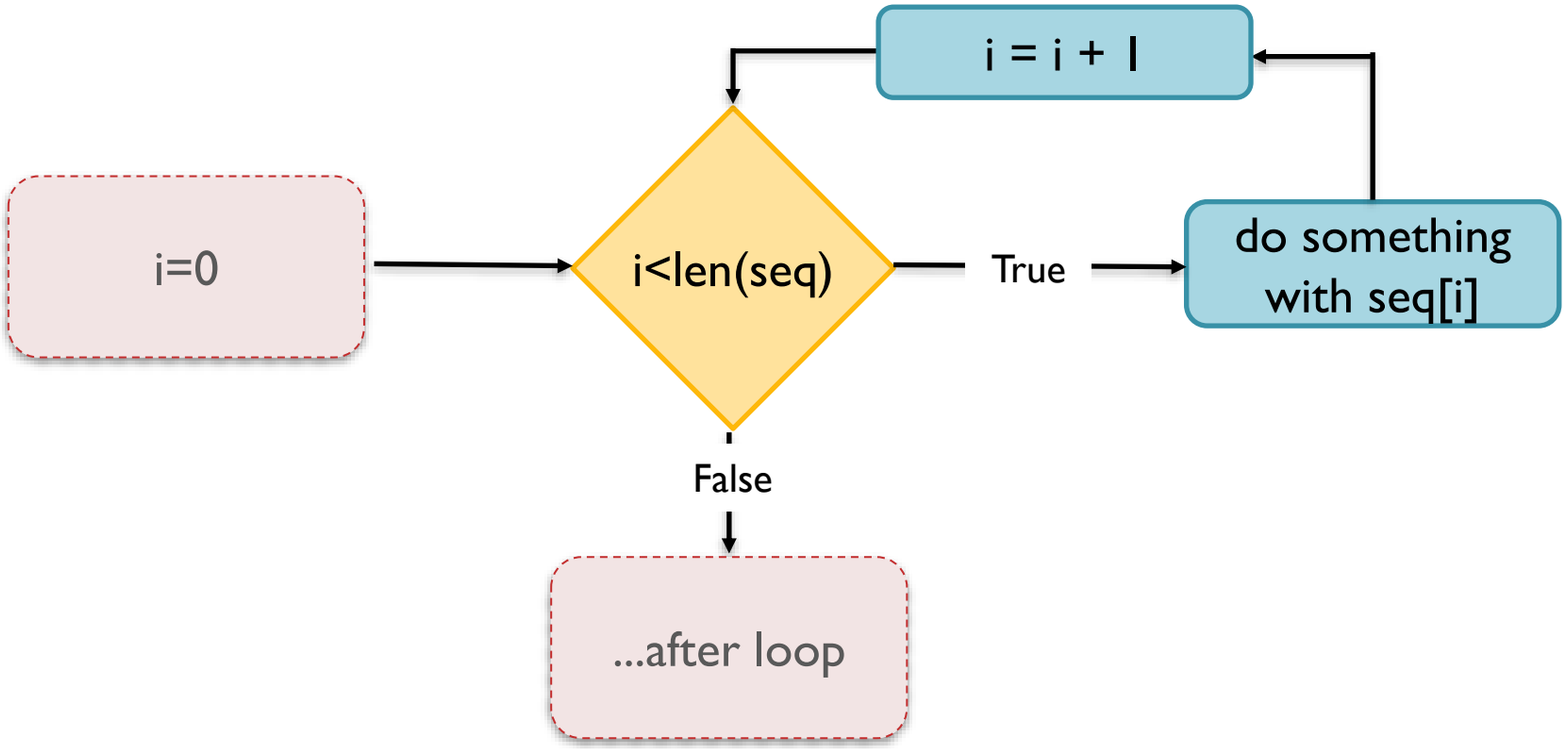
```
def times_eaten(food, eaten_foods):
    """
    Input : specific food, list of eaten foods
    Output: number of times food appears in eaten_foods"""
```



# We know how to solve problem with while-loop

```
def times_eaten(food, eaten_foods):
    """
    Input : specific food, list of eaten foods
    Output: number of times food appears in eaten_foods"""
    i = 0
    res = 0
    while i < len(eaten_foods):
        if eaten_foods[i] == food:
            res = res + 1
        i = i + 1
    return res
```

requires manual index "bookkeeping"



# Problem: check whether collections have element in common

```
def violated_diet(eaten_foods, forbidden_foods):
    """
    Input : collections of eaten foods and foods forbidden in diet
    Output: whether diet has been violated
    """
    i = 0
    while i < len(eaten_foods):
        if eaten_foods[i] in forbidden_foods:
            return True
        i = i + 1
    return False
```

```
>>> eaten = ['milk', 'eggs', 'bread', 'jam', 'bread']
>>> low_carb = {'fries', 'bread', 'jam'}
>>> vegan = {'meat', 'fish', 'eggs', 'milk'}
>>> vegetarian = {'meat', 'fish'}
>>> violated_diet(eaten, low_carb)
True
>>> violated_diet(eaten, vegan)
True
>>> violated_diet(eaten, vegetarian)
False
```

# Indexing approach does not work for non-sequence collections

```
def violated_diet(eaten_foods, forbidden_foods):
    """
    Input : collections of eaten foods and foods forbidden in diet
    Output: whether diet has been violated
    """
    i = 0
    while i < len(eaten_foods):
        if eaten_foods[i] in forbidden_foods:
            return True
        i = i + 1
    return False
```

now specify eaten foods as set

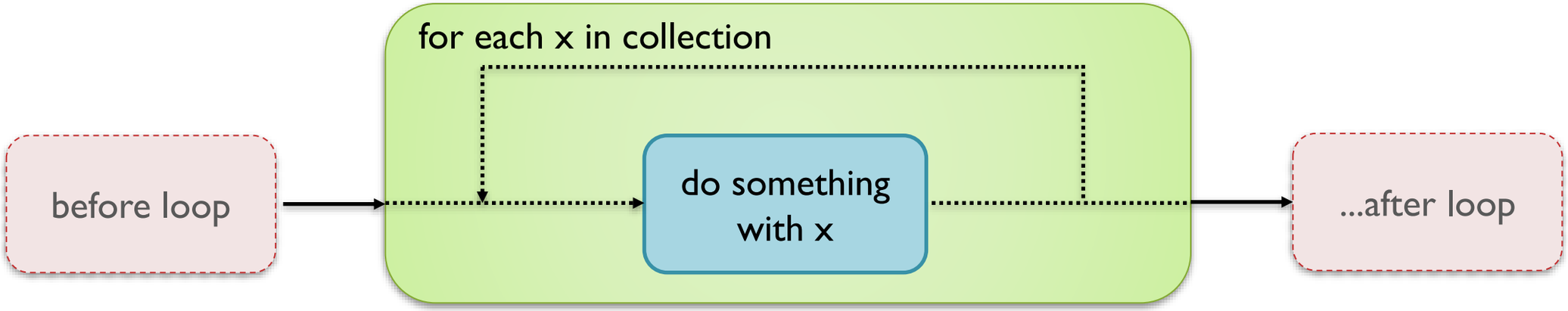
```
>>> eaten = {'milk', 'eggs', 'bread', 'jam', 'bread'}
>>> low_carb = {'fries', 'bread', 'jam'}
>>> vegan = {'meat', 'fish', 'eggs', 'milk'}
>>> vegetarian = {'meat', 'fish'}
>>> violated_diet(eaten, low_carb)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

https://flux.qa

Clayton: AXXULH
Malaysia: LWERDE

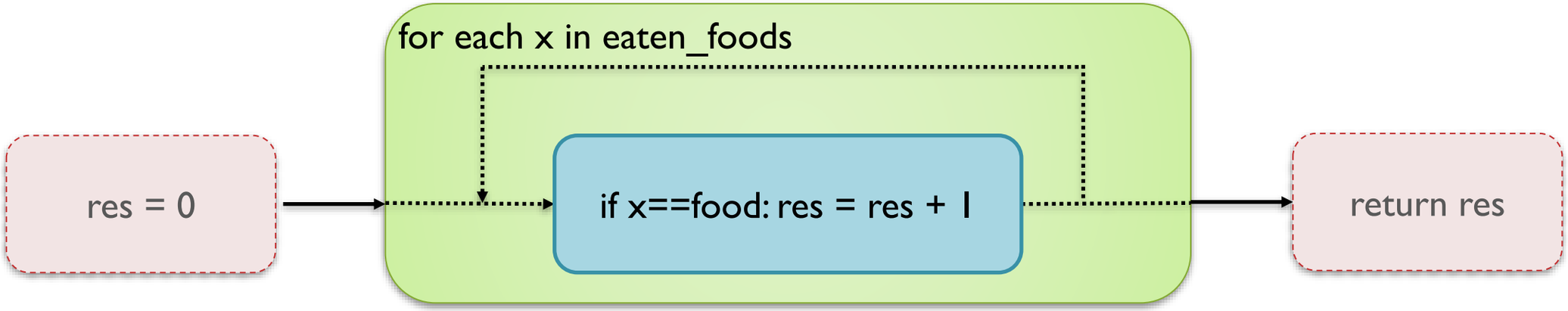
How to even investigate elements of arbitrary collection?

# Want loop that runs *once per element* w/o bookkeeping



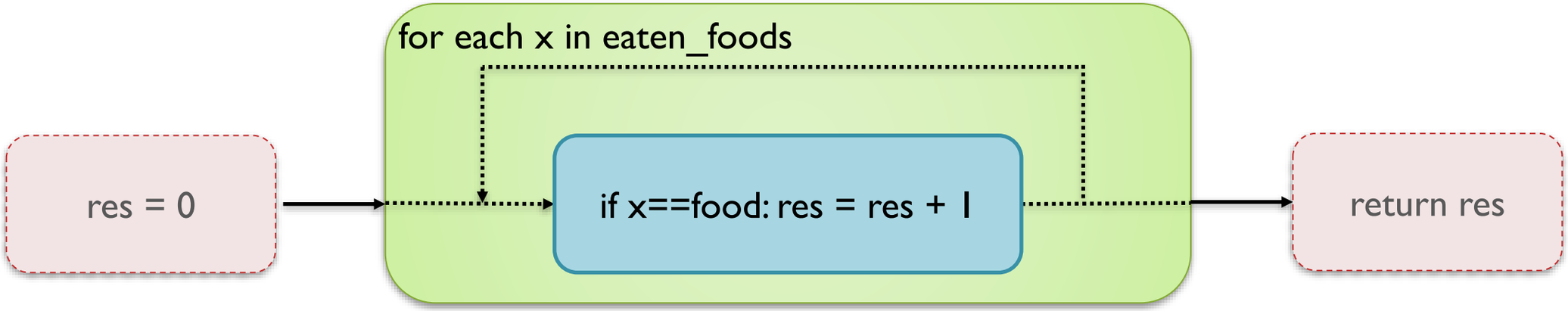
# Want loop that runs once per element w/o bookkeeping

```
def times_eaten(food, eaten_foods):
    """
    Input : specific food, list of eaten foods
    Output: number of times food appears in eaten_foods
    """
```



# This is what Python for-loops do with the usual indentation syntax

```
def times_eaten(food, eaten_foods):
    """
    Input : specific food, list of eaten foods
    Output: number of times food appears in eaten_foods
    """
    res = 0
    for f in eaten_foods:
        if f == food: res = res + 1
    return res
```



# For-loop works with arbitrary collections

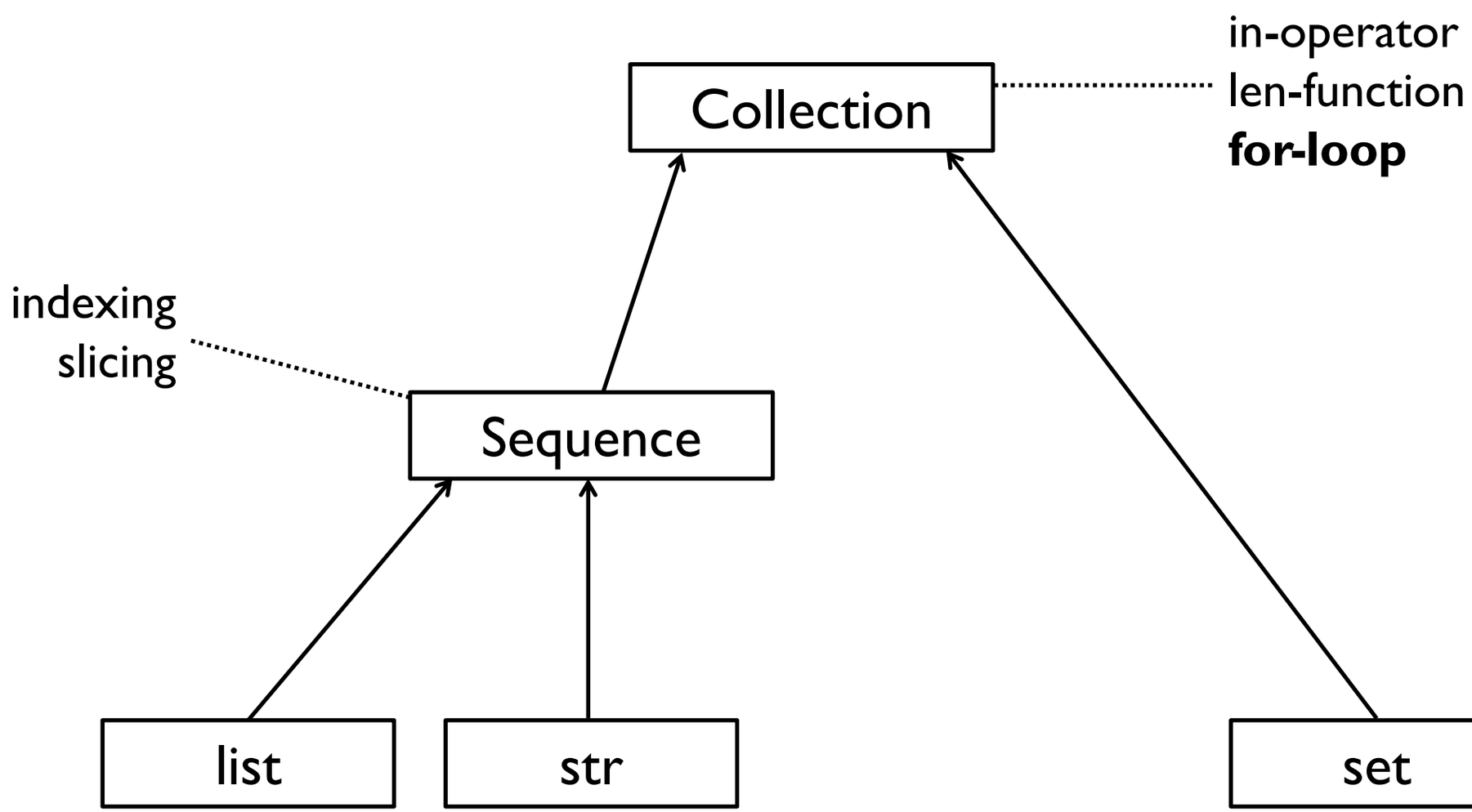
```
def violated_diet(eaten_foods, forbidden_foods):
    for food in eaten_foods:
        if food in forbidden_foods: return True
    return False
```

comparison between loop types

```
def violated_diet(eaten_foods, forbidden_foods):
    i = 0
    while i < len(eaten_foods):
        if eaten_foods[i] in forbidden_foods: return True
        i = i + 1
    return False
```

```
>>> eaten = {'milk', 'eggs', 'bread', 'jam', 'bread'}
>>> low_carb = {'fries', 'bread', 'jam'}
>>> vegan = {'meat', 'fish', 'eggs', 'milk'}
>>> vegetarian = {'meat', 'fish'}
>>> violated_diet(eaten, low_carb)
True
>>> violated_diet(eaten, vegan)
True
>>> violated_diet(eaten, vegetarian)
False
```

# Collections type hierarchy



# Where am I?

1. Collections
2. For-loops
3. Ranges

# Often we need to link data via index

```
def quantity_eaten(food, eaten_foods, eaten_quantities):
    """
    Input : specific food, list of eaten foods,
            list of eaten quantities
    Output: number of times food appears in eaten_foods
    """
```

```
>>> eaten = ['beef', 'potato', 'broccoli', 'apple', 'potato',
'apple']
>>> quantities = [300, 300, 200, 100, 250, 100]
>>> quantity_eaten('apple', eaten, quantities)
200
```

# Let's try to use our shiny new tool

```
def quantity_eaten(food, eaten_foods, eaten_quantities):
    """
    Input : specific food, list of eaten foods,
            list of eaten quantities
    Output: number of times food appears in eaten_foods
    """
    res = 0

    for f in eaten_foods:
        if f == food:
            res = res + ?

    return res
```

how to find matching quantity for food f?

```
>>> eaten = ['beef', 'potato', 'broccoli', 'apple', 'potato',
'apple']
>>> quantities = [300, 300, 200, 100, 250, 100]
>>> quantity_eaten('apple', eaten, quantities)
200
```

# We need to have access to index

```
def quantity_eaten(food, eaten_foods, eaten_quantities):
    """
    Input : specific food, list of eaten foods,
            list of eaten quantities
    Output: number of times food appears in eaten_foods
    """
    res = 0
    i = 0
    for f in eaten_foods:
        if f == food:
            res = res + eaten_quantities[i]
        i = i + 1
    return res
```

solution with index bookkeeping brings us back to while-loop

```
>>> eaten = ['beef', 'potato', 'broccoli', 'apple', 'potato',
'apple']
>>> quantities = [300, 300, 200, 100, 250, 100]
>>> quantity_eaten('apple', eaten, quantities)
200
```

Feels like for-loops are rather limited

… until you learn about ranges ☺

# *Ranges*: sequences of consecutive integers

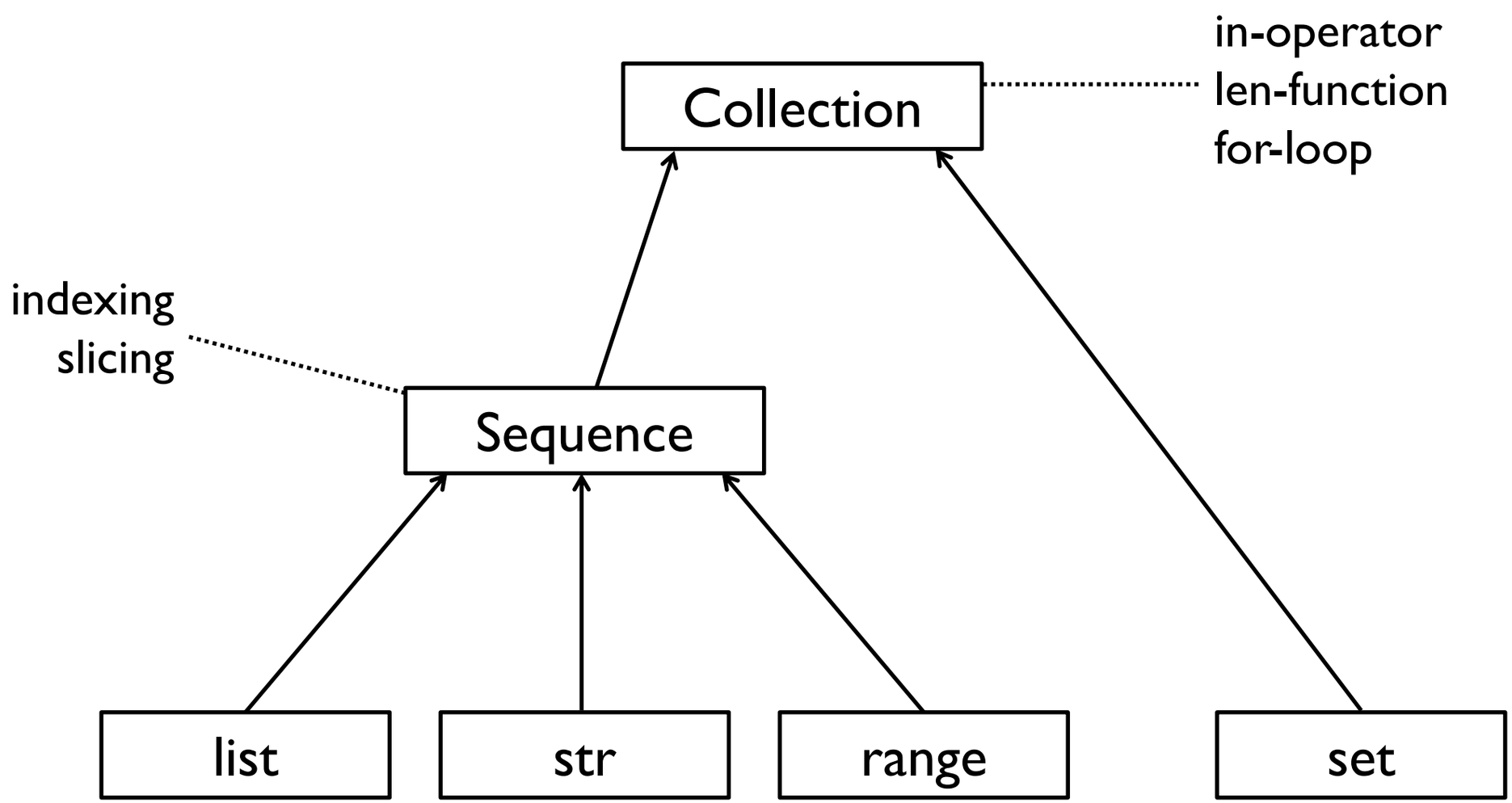
```
>>> indices = range(8,15)
>>> indices
range(8, 15)
>>> type(indices)
<class 'range'>
>>> 8 in indices
True
>>> 15 in indices
False
>>> len(indices)
7
>>> list(indices)
[8, 9, 10, 11, 12, 13, 14]
>>> indices[2]
10
>>> indices[2:]
range(10, 15)
```

stop integer exclusive

start integer inclusive

- Range objects represent *sequences* of consecutive integers
- Specific purpose: to be used with for-loops

# Collections type hierarchy



# For-loop over index solves problem without bookkeeping

```
def quantity_eaten(food, eaten_foods, eaten_quantities):
    """
    Input : specific food, list of eaten foods,
            list of eaten quantities
    Output: total quantity of specific food eaten
    """
    res = 0
    for i in range(0, len(eaten_foods)):
        if eaten_foods[i] == food:
            res = res + eaten_quantities[i]
    return res
```

start index 0 is so common that it can be omitted

# For-loop over index solves problem without bookkeeping

```
def quantity_eaten(food, eaten_foods, eaten_quantities):
    """
    Input : specific food, list of eaten foods,
            list of eaten quantities
    Output: total quantity of specific food eaten
    """
    res = 0
    for i in range(len(eaten_foods)):
        if eaten_foods[i] == food:
            res = res + eaten_quantities[i]
    return res
```

comparison between loop types

```
def quantity_eaten(food, eaten_foods, eaten_quantities):
    res = 0
    i = 0
    while i < len(eaten_foods):
        if eaten_foods[i] == food:
            res = res + eaten_quantities[i]
        i = i + 1
    return res
```

# Code complexity adds up quickly

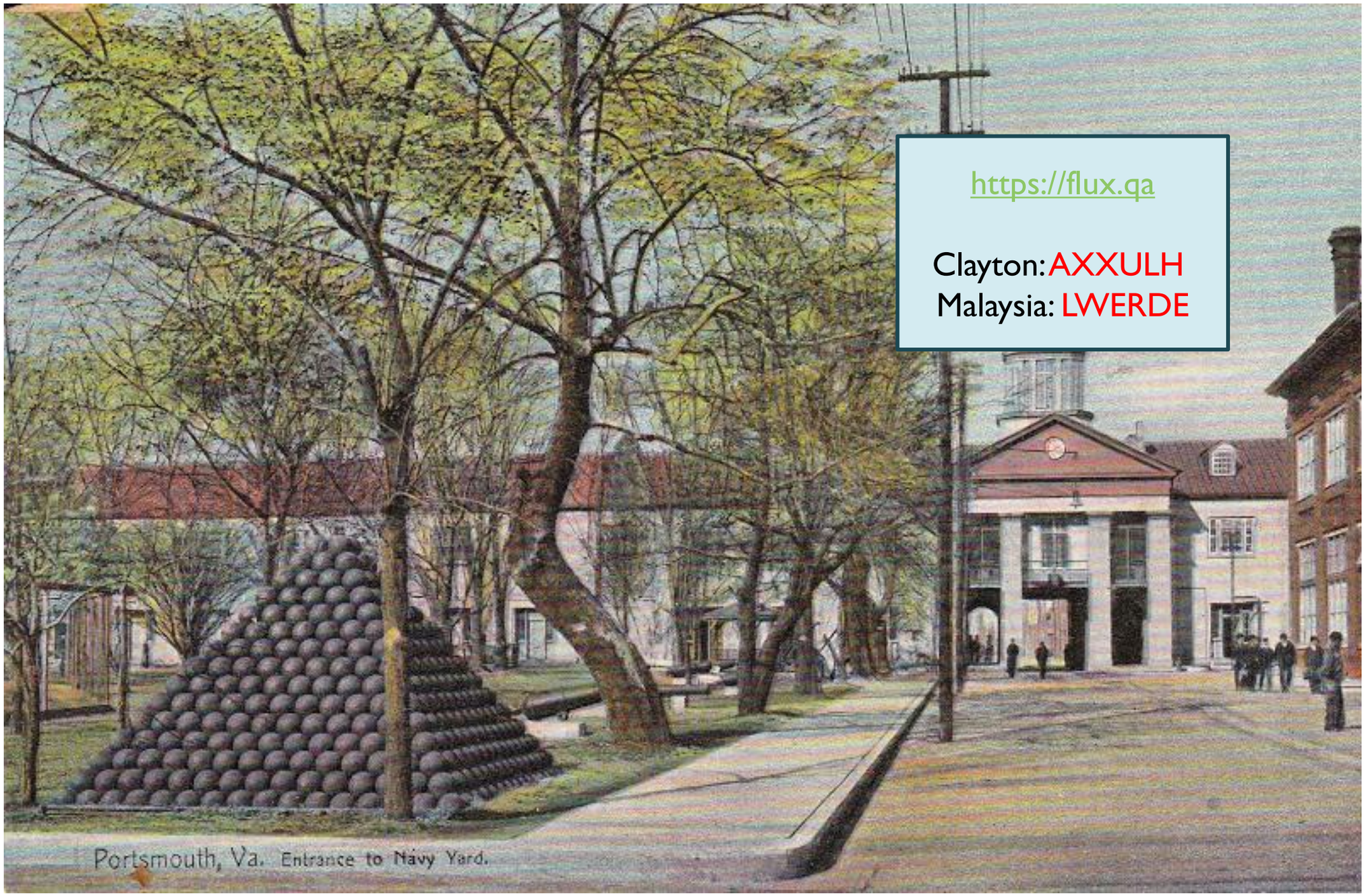
```
def have_common_element(s1, s2):
    for a in s1:
        for b in s2:
            if a==b:
                return True
    return False
```

comparison between loop types

```
def have_common_element(s1, s2):
    i = 0
    while i < len(s1):
        j = 0
        while j < len(s2):
            if s1[i]==s2[j]:
                return True
            j = j + 1
        i = i + 1
    return False
```

Python is designed to keep it simple!

# How many cannonballs are there in the pile?




https://flux.qa

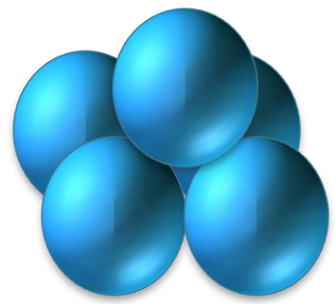
Clayton: AXXULH
Malaysia: LWERDE

http://www.nnsyhistorymatters.com/2015/02/the-cannonball-pile-at-trophy-park.html

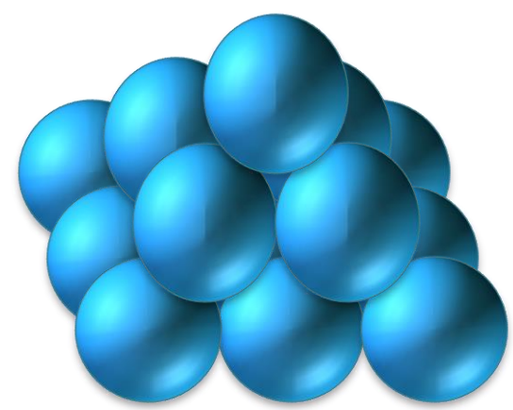
# Square Pyramidal Numbers

$1$

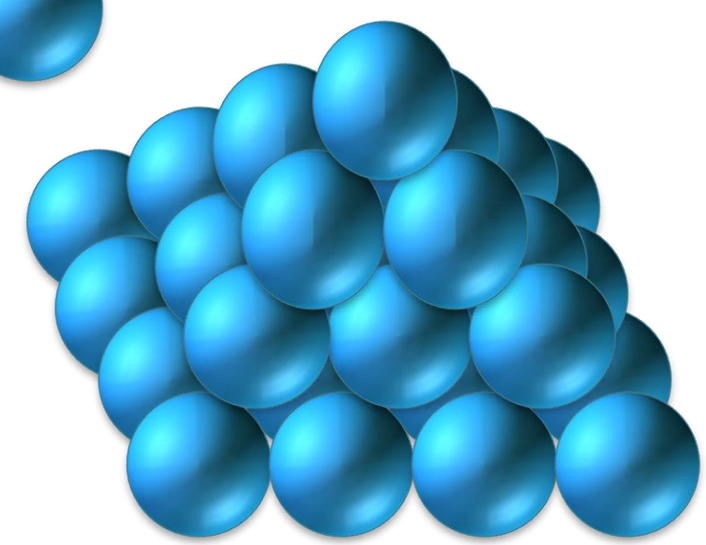
$1^2 + 2^2$



$1^2 + 2^2 + 3^2$



$1^2 + 2^2 + 3^2 + 4^2$



# Problem: compute the *n*-th square pyramidal number

```
def pyramidal(n):
    """
    Input: an integer n
    Output: number of cannonballs in pile of height n
    """
```

# Problem: compute the *n*-th square pyramidal number

```
def pyramidal(n):
    """
    Input: an integer n
    Output: number of cannonballs in pile of height n
    """
    count = 0
```

# Problem: compute the *n*-th square pyramidal number

```
def pyramidal(n):
    """
    Input: an integer n
    Output: number of cannonballs in pile of height n
    """
    count = 0
    for k in range(1, n+1):
```

# Problem: compute the *n*-th square pyramidal number

```
def pyramidal(n):
    """
    Input: an integer n
    Output: number of cannonballs in pile of height n
    """
    count = 0
    for k in range(1, n+1):
        count = count + k**2
```

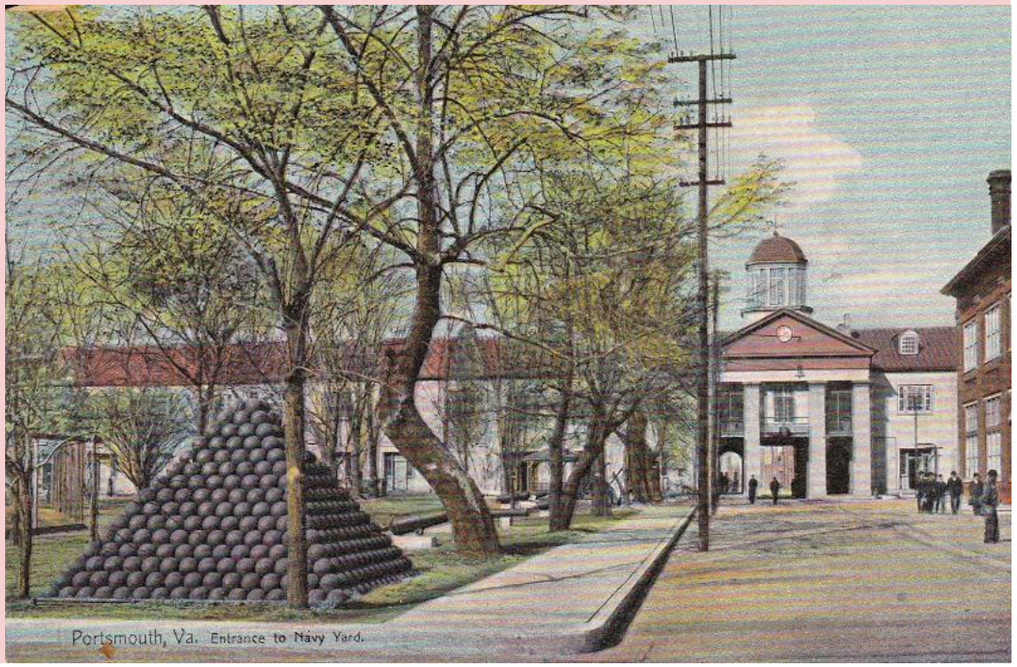
# Problem: compute the *n*-th square pyramidal number

```
def pyramidal(n):
    """
    Input: an integer n
    Output: number of cannonballs in pile of height n
    """
    count = 0
    for k in range(1, n+1):
        count = count + k**2
    return count
```

# Problem: compute the *n*-th square pyramidal number

```
def pyramidal(n):
    """
    Input: an integer n
    Output: number of cannonballs in pile of height n

    For example:
    >>> pyramidal(1)
    1
    >>> pyramidal(3)
    14
    """
    count = 0
    for k in range(1, n+1):
        count = count + k**2
    return count
```




```
>>> pyramidal(16)
1496
```

# Advanced: *step* parameter in slices and ranges

```
>>> items = ['milk', 'eggs', 'bread', 'jam', 'bread']
>>> items[0:len(items):2]
['milk', 'bread', 'bread']
>>> range(1, 20, 4)
range(1, 20, 4)
>>> list(range(1, 20, 4))
[1, 5, 9, 13, 17]
>>>
```

Resulting ranges and slices skip over elements.

Can you use this to create decreasing ranges?

# Summary

- Sequences are ordered collections of objects that allow indexing and slicing
- There are other collection objects that only allow membership testing and length determination
- For-loops allow to "loop over" all elements in arbitrary collection without "index bookkeeping"
- Ranges are special sequence objects for representing relevant index collections

# Recommended reading

"*Introduction to Computing using Python: An Application Development Focus*", by L. Perkovic

- **Sections 2.3 and 5.3**

**FIT1045/53 Workbook**

- Chapter 2, § 2.2.2
- Chapter 3, § § 3.1-3.3

# On Wednesday

- Organising data in tables
- Reading and writing data from files

| food | energy | water | protein | carbs | sugars | fat | fibres |
|---|---|---|---|---|---|---|---|
| apple | 229 | 84.3 | 0.4 | 12 | 11.8 | 0 | 2.3 |
| orange | 186 | 84.3 | 1 | 9.5 | 8.3 | 0.2 | 2.1 |
| broccoli | 124 | 89.6 | 3.2 | 2 | 2 | 0.1 | 4.1 |
| beef | 613 | 70 | 22.8 | 0.2 | 0 | 6 | 0 |
| lamb | 1057 | 60.2 | 18.6 | 0 | 0 | 20.2 | 0 |
| bread | 1446 | 37.6 | 8.4 | 43.5 | 1.5 | 2.6 | 6.9 |
| potato | 346 | 77.4 | 2 | 17 | 0 | 0.1 | 2.5 |
| tofu | 510 | 74 | 12 | 1.5 | 0.5 | 6.5 | 5 |
| tomato | 81 | 93.3 | 1 | 2.9 | 0.9 | 0.2 | 1 |
| eggplant | 107 | 91.6 | 1.2 | 3.5 | 1.5 | 0.2 | 2.5 |
| carrot | 116 | 90.6 | 0.8 | 4.7 | 4.4 | 0 | 2.9 |
| coco. cream | 872 | 73 | 1.5 | 3 | 0 | 21.5 | 0 |
| rice | 403 | 75.3 | 2.5 | 20 | 0 | 0.4 | 0.8 |

database of nutrition values

| id | day | food | quantity |
|---|---|---|---|
| 1 | 1 | beef | 300 |
| 2 | 1 | potato | 300 |
| 3 | 1 | broccoli | 200 |
| 4 | 1 | apple | 100 |
| 5 | 2 | potato | 250 |
| 6 | 2 | apple | 100 |
| 7 | 2 | tofu | 120 |
| 8 | 2 | tomato | 200 |
| 9 | 3 | rice | 220 |
| 10 | 3 | carrot | 120 |
| 11 | 3 | eggplant | 150 |
| 12 | 3 | coconut cream | 160 |
| 13 | 3 | apple | 110 |

food diary