Tuples, sets, and dictionaries

1DV501/1DT901 - Introduction to Programming

Jonas Lundberg, office B3024

Jonas.Lundberg@lnu.se

Lecture slides are available in Moodle

September 30, 2020

Today ...

- Data structures in general
- Tuples
- Sets
- Dictionaries
- Various left-overs
- Mini-project information (Not completed)

Reading instructions:

Data Structures – Introduction

- We often need to handle large sets of data
- A *data structure* is a model for storing/handling such data sets
- Scenarios where data structures are needed
  1. Students in a course
  2. Measurements from an experiment
  3. Queue to get an apartment at our campus
  4. Telephone numbers in Stockholm
- Different scenarios require different data structure properties
  - Data should be ordered
  - Not the same element twice
  - Data come in pairs or have relations between them
  - Important that look-up is fast
  - In general: Important that operations X,Y,Z are fast
- Selecting data structure is a design decision $\Rightarrow$ might affect performance, modifiability, and program comprehension.

A Few Common Data Structures

List A *sequential* collection where each *element* has a position. In principal: a growing/flexible sequence of data

Queue A sequential collection with add and remove at different sides ⇒ a *FiFo* (First in, First out)

Stack A sequential collection with add and remove at the same side ⇒ a *LiFo* (Last in, First out)

Deque A sequential collection with add and remove at both sides (Deque = Double-Ended Queue)

Set A non-ordered collection not containing the same element twice ⇒ Trying to add `X` twice ⇒ the second attempt is ignored

Map (or *Table* or *Dictionary*) A set of *key/value* pairs
Operations: `put(key,value)`, `get(key) --> value`

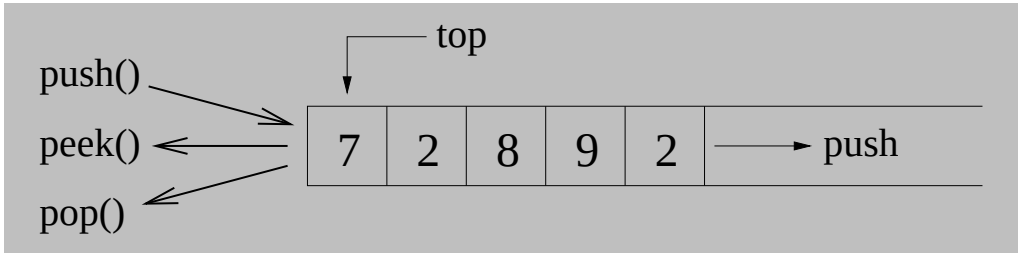
Tree Data ordered as a tree with a root (Will be presented later)
Example: The file system on your hard drive

Graph Data (nodes) with binary relations (edges)
Example: A road map with cities (nodes) and roads (edges) between them. Not part of this course.

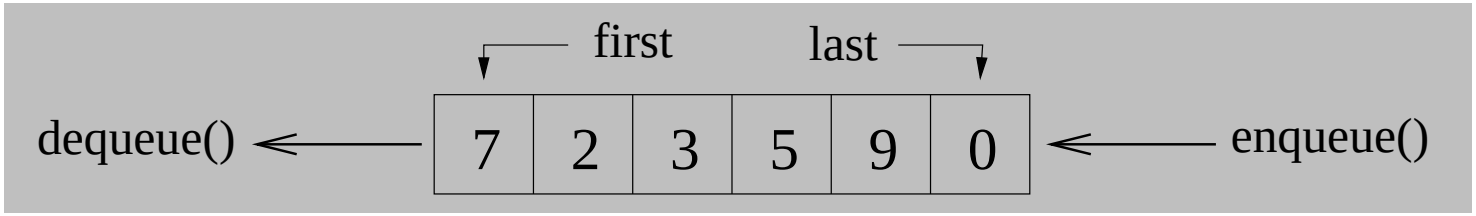
Sequential Data Structures

Sequential ⇒ a sequence where each element has a position.

Stack (Last in, first out) Add and remove at one side only.



Queue (First in, first out) Add at one side, remove at the other.

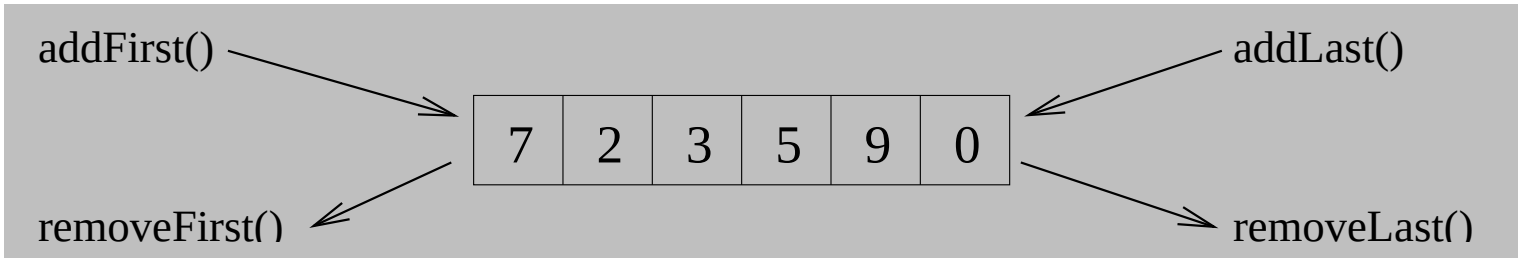


Note: Stack and Queue has a very limited set of operations

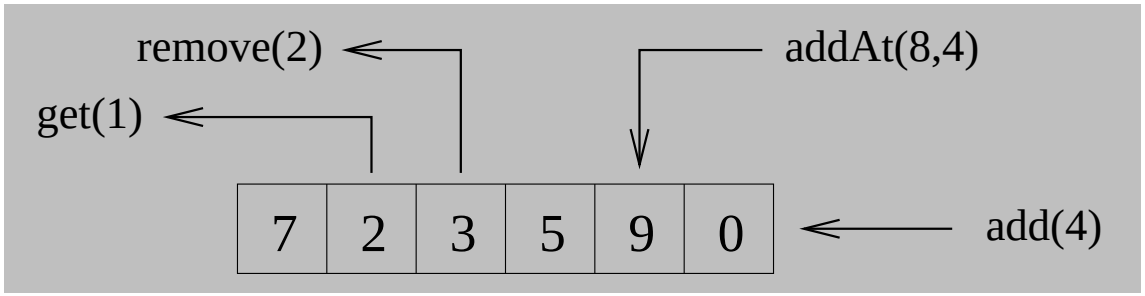
⇒ they are the most simple data structures

Deque and List

Deque (Double-ended Queue) Add and remove at both sides.

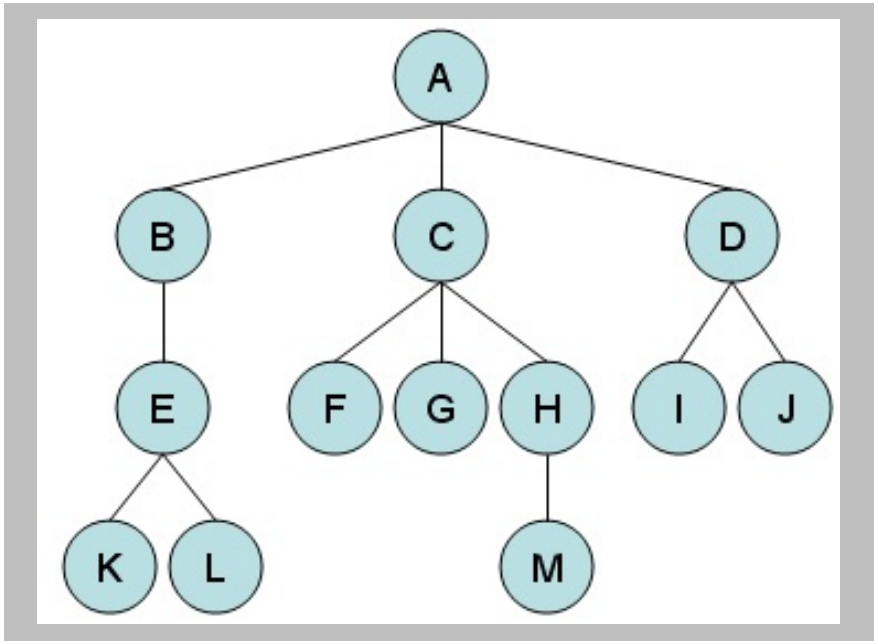


List (Add and remove everywhere)



- Note: List is the most general sequential data structure
- Q: Why not always use a list?
  - A specialized data structure can be more efficient (time, memory).
  - A specialized structure provides a more precise model
    $\Rightarrow$ easier to understand for someone who reads the code

Tree

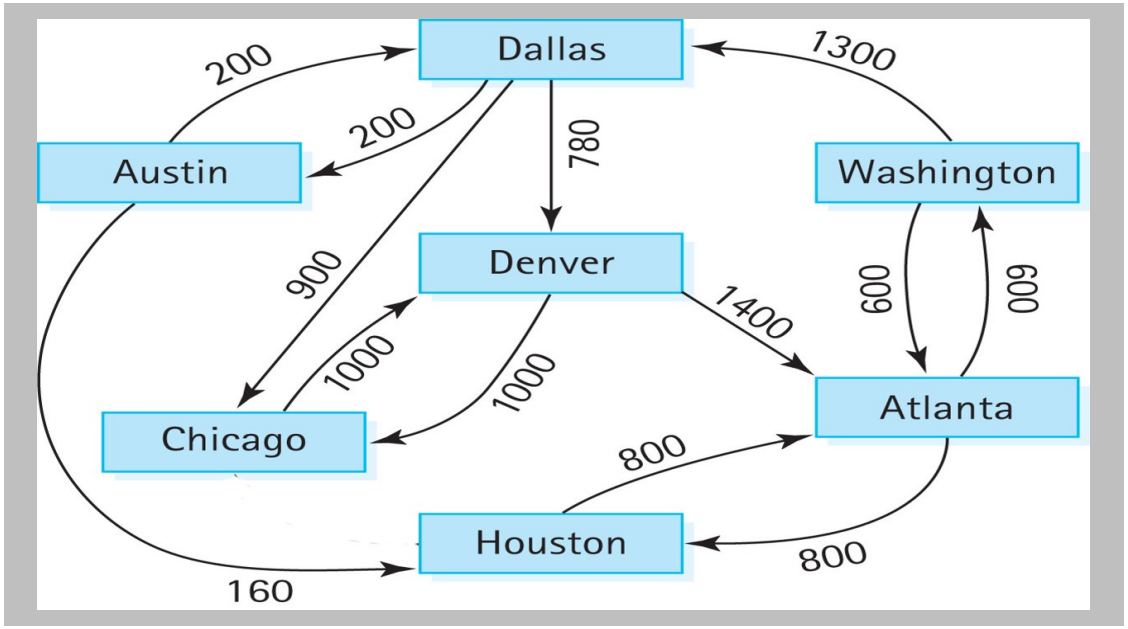


Tree

- A unique node (A) called the **root** with no parent node
- All other nodes have one **parent** and zero or more **children**
- Nodes with no children (e.g. K,L, F) are called **leafs**

Trees are very good at organizing things that are hierarchical. For example, directories on our hard drive.

Graph



Graph

- A set of **nodes** connected with **edges**
- All nodes can connect to any other nodes
- Edges might have **labels** adding information to the connection

Graphs are very good at displaying networks, or any data set where the different data points (nodes) are connected with each other.

Data Structures – Definition

A data structure is defined by:

1. a name
2. the type of data that can be stored
3. a number of operation definitions

Note: What type of implementation that is used is not a part of the definition.
Example:

- Name: StringStack
- Data type: Strings
- Operations
  - push: Add a string at the top of the stack
  - pop: Return (and remove) the string at the top of the stack
  - peek: Return (without removing) the string at the top of the stack
  - size: Returns the current number of strings in the stack
  - ...

  Note: This is more of a *description* rather than a formal definition. We can use mathematics (so-called formal specifications) to properly define the semantics of each operation.

Introduction to tuples

```
tpl = (1,2,3,4,5)      # Create a new tuple
print(tpl, type(tpl)) # (1, 2, 3, 4, 5) <class 'tuple'>

a = tpl[2]             # Access element at position 2
print(a, type(a))      # 3 <class 'int'>

b = tpl[1:3]           # Slice elements at positions 2 and 3
print(b, type(b))      # (2, 3) <class 'tuple'>

for n in tpl:          # Iterate over all tuple element
    print(n, end=" ")
print()                # 1 2 3 4 5
```

- Tuples are a sequential data structure
- Tuples are created using parentheses like `(1,2,3,4,5)`
- Elements can be accessed individually (e.g. `tpl[2]`) or as slices (e.g. `tpl[1:3]`)
- Easy to iterate over all elements: `for n in tpl: ...`

Creating new tuples

```
odd = (1,3,5)      # Create a new tuple object
even = (2,4,6)
zero = 3*(0,0)     # Tuple multiplication

tpl = odd+even+zero  # Tuple concatenation
print(tpl)           # (1, 3, 5, 2, 4, 6, 0, 0, 0, 0, 0, 0)

one = (1)
two = (2,)           # Notice comma inside parentheses
three = 3*(1,)
empty = ()
print(one, type(one))     # 1 <class 'int'>
print(two, type(two))     # (2,) <class 'tuple'>
print(three,type(three)) # (1, 1, 1) <class 'tuple'>
print(empty,type(empty)) # () <class 'tuple'>
```

- Tuples can be concatenated and multiplied to form new tuples
- `()` is the empty tuple and `(2,)` is tuple with one element
- **Notice**: You need a comma to create tuples with only one element, Python interprets `(1)` as the integer 1.

Tuples are immutable

Immutable ⇒ can not be modified once created

```
tpl = (1,2,3,4,5)      # Create a new tuple

tpl[2] = 99     # Error since immutable
del tpl[2]      # Error since immutable
```

Tuples have only two methods

```
tpl = (2,2,5,5,5)      # Create a new tuple

print( tpl.count(5))  # 3, Number of times 5 occurs in tuple
print( tpl.index(5))  # 2, First position of 5
```

Heterogeneous data

```
tpl = (2,(5,6,7), "Hello", [89,10])
```

Tuples (like lists) can hold heterogeneous data types

Lists vs Tuples

Table 11.1 Python lists versus tuples

Feature	List	Tuple
Mutability	mutable	immutable
Creation	`lst = [i, j]`	`tpl = (i, j)`
Element access	`a = lst[i]`	`a = tpl[i]`
Element modification	`lst[i] = a`	*Not possible*
Element addition	`lst += [a]`	*Not possible*
Element removal	`del lst[i]`	*Not possible*
Slicing	`lst[i:j:k]`	`tpl[i:j:k]`
Slice assignment	`lst[i:j] = []`	*Not possible*
Iteration	`for elem in lst:...`	`for elem in tpl:...`

Tuples are a lighweight version of lists. The most significant difference is that tuples are immutable.

Convert between tuple and list

Convert between tuple and list is easy using `list()` and `tuple()`

```
lst = list( range(1,6) )          # Range to list
tpl = tuple(lst)                  # List to tuple
print(lst, tpl)     # [1, 2, 3, 4, 5] (1, 2, 3, 4, 5)

tpl = tuple( range(10,51,10) )  # Range to tuple
lst = list( tpl)                  # Tuple to list
print(lst, tpl)     # [10, 20, 30, 40, 50] (10, 20, 30, 40, 50)
```

Built-in functions work as expected for tuples

```
tpl = tuple( range(5) )
print(tpl)           # (0, 1, 2, 3, 4)
print( len(tpl) )    # 5
print( min(tpl) )    # 0
print( max(tpl) )    # 4
print( sum(tpl) )    # 10
```

Unpacking tuples

Unpacking ⇒ assign tuple content to variables.

```
tpl = (1,2,3)       # Create tuple
a, b, c = tpl       # Unpacking tuple
print(a,b,c)        # Output: 1 2 3
```

Functions returning multiple values are actually returning tuples.

```
def m():  # Function returning three values
    x, y, z = 5, 6, 7
    return x,y,z

# Program starts
p = m()               # Receive multiple values as a tuple
print(p, type(p))     # (5, 6, 7) <class 'tuple'>

x,y,z = m()           # Unpack received tuple
print(x,y,z)          # Output: 5 6 7
```

Hence, tuples and multi-assignments are closely connected

Tuples - A poor man's lists?

Why bother with tuples when we have lists? After all, lists can do everything tuples can and much more.

Tuple advantages

- Tuple are immutable $\Rightarrow$ less error prone
- Tuple access (e.g. `a = tpl[6]`) is faster
- Tuple creation is faster
- Tuple iteration is faster
- Tuples consume less memory

Hence, use tuples if possible for large data sets and time consuming operations.

Recommendation: Use tuples when you know the sequence size in advance and have no need for updating the sequence.

Introduction to Sets

Set: A non-sequential data structure with no duplicate elements

```
st = {4,3,2,1,2,3,4}
print("Set content:", st)    # Set content: {1, 2, 3, 4}
```

Notice: All duplicate elements are removed

Starting from an empty set

```
import random as rnd

st = set()                   # New empty set
for i in range(10):          # Ten random integers in range 1 to 10
    r = rnd.randint(1,10)
    st.add(r)                # Add if not already in set
print(len(st), st)           # 6 {1, 3, 4, 5, 7, 9}
```

- Non empty sets are created using curly brackets (e.g. `{1,2,3}`)
- An empty set is created using `set()` (not `{}`, `{}` is reserved for dictionaries)
- Sets are mutable $\Rightarrow$ we can add more elements using `st.add(r)`

Sets: No element order

```
strings = {"Hello", "Hej", "Hola", "Ciao", "Hej"}
print(strings)   # {'Hej', 'Ciao', 'Hello', 'Hola'}

floats = {4.17, 2.3, -1.1, 2.3}
print(floats)    # {2.3, 4.17, -1.1}

ints1 = {3, 17, -1, 6, 3}
print(ints1)     # {17, 3, -1, 6}

ints2 = {5,4,3,99,1,6}
print(ints2)     # {1, 99, 3, 4, 5, 6}
```

Notice: Output in no particular order $\Rightarrow$ sets are not ordered

```
strings = {"Hello", "Hej", "Hola", "Ciao", "Hej"}
for s in strings:
    print(s, end=" ")  # Output: Hola Ciao Hej Hello
print()
```

Set iteration similar to list, tuple, and string iteration (but in arbitrary order)

Sets: Non-sequential

Sets are non-sequential and non-ordered $\Rightarrow$ elements have no positions, no indexing and, no slicing

```
ints = {1,3,5,7,9,11}

a = ints[2]          # Error
sub = ints[1:3]      # Error
```

Non-sequential

- No positions, no indexing (e.g. `ints[2]`) and, no slicing (e.g. `ints[1:3]`
- No first element, no last element
- Iteration order is arbitrary

Sets operations

Set is a class ⇒ comes with many methods

```
ints = {1,3,5,7}

ints.add(8)      # Add 8 if not already in set

if 5 in ints:    # Check if 5 is in set
    print("5 is in the list")

ints.update({2,4,6,8}) # Add a set of elements
print(ints)      # {1, 2, 3, 4, 5, 6, 7, 8}

ints.remove(5)     # Removes 5 from set, KeyError if not present
ints.discard(10)   # Removes x from set if present
print(ints)        # {1, 2, 3, 4, 6, 7, 8}

cp = ints.copy()   # New set with same elements as ints
ints.clear()       # Remove all elements from ints
print(ints, cp)    # set() {1, 2, 3, 4, 6, 7, 8}}
```

Sets as mathematical set

Python sets support mathematical set operations like union, intersection, ...

- `s.issubset(t)`: test whether every element in s is in t $\Rightarrow$ boolean result
- `s.issuperset(t)`: test whether every element in t is in s $\Rightarrow$ boolean result
- `s.union(t)`: new set with elements from both s and t
- `s.intersection(t)`: new set with elements common to s and t
- `s.difference(t)`: new set with elements in s but not in t
- `s.symmetric_difference(t)`: new set with elements in either s or t but not both

Not part of this course, but feel free to use them if you are comfortable with mathematical set operations

Python Set - Summary

Sets in Python are a non-sequential and non-ordered data structure with no duplicate elements

- Empty sets are created using `set()`
- Non-empty sets are created using curly brackets like `{3,1,2}`
- Sets are mutable ⇒ we can add and remove elements using set methods
- Non-ordered ⇒ arbitrary iteration and print-out order
- Set iteration is simple: `for s in myset: ...`
- Sets support mathematical set operations like union and intersection

When to use sets

- No duplicate elements is sometimes a very useful property
- Count number of different X ⇒ add all X to a set and check the set size
- Looking up an element (i.e. `x in myset`) is much faster for sets compared to lists or tuples. Implementing sets will be a part of the mini-project.

Introduction to Dictionaries

```
# Create dictionary
dct = {"Jonas":56, "Ola": 55, "Tobias":43, "Morgan":45, "Fredrik":36}

age = dct["Ola"]                    # Look up value for key "Ola"
print("Age of Ola:", age)

dct["Jonas"] = 26                   # Assign "Jonas" a new value
print(dct)                          # Print dictionary content
```

Output

```
Age of Ola: 55
{'Jonas': 26, 'Ola': 55, 'Tobias': 43, 'Morgan': 45, 'Fredrik': 36}
```

- A **dictionary** in Python is set of **key-value** pairs
- Often called **map** or **table** in other programming languages
- Here (`'Jonas': 56`) and (`'Ola': 55`) are two key-value pairs
- `age = dic["Ola"]` $\Rightarrow$ We look up the value corresponding to key `"Ola"`
- A dictionary is designed for a speedy look up of the value for a certain key.

Dictionaries: add and look-up

```
# Create start dictionary
dct = {"Jonas":56, "Ola": 55, "Tobias":43, "Morgan":45, "Fredrik":36}

dct["Jesper"] = 50          # Add new key-value pair
dct["Jonas"] = 57           # Assign key "Jonas" a new value

old = dct["Jonas"]          # Look up value for "Jonas" ==> OK
young = dct["Simon"]        # Look up non-existing key ==> KeyError
```

- `dct["Jesper"] = 50` ⇒ Add new key-value pair since "Jesper" is a new key
- `dct["Jonas"] = 57` ⇒ Assign a new value since "Jonas" is an existing key
- `old = dct["Jonas"]` ⇒ Look up value for "Jonas" ⇒ OK since an existing key
- `young = dct["Simon"]` ⇒ KeyError since key "Simon" is missing

Starting from an empty dictionary

```
dct = {}   # An empty dictionary

for i in range(10,16):  # 10, 11, 12, 13, 14, 15
    sq = i*i
    dct[i] = sq       # Add new key-value pair
print(dct)   # {10: 100, 11: 121, 12: 144, 13: 169, 14: 196, 15: 225}
print("Dictionary size:", len(dct))  # Dictionary size: 6
```

- `dct = {}` ⇒ We create a new empty dictionary
  Hence, `{}` is an empty dictionary and `set()` is an empty
- Dictionary can handle all types of keys and values
- Heterogeneous types:
  `{8: 44, True: right, Beta: 100, 3.4: True, Alpha: up}`
- `len(dct)` ⇒ Dictionary size ⇒ Number of key-value pairs

Iterating over dictionary pairs

```
d = {"Fred" : 44, "Ella" : 39, "Owen" : 40, "Zoe" : 41}

for k in d.keys():          # Iterate over keys
    print(k, end= " ")      # Fred Ella Owen Zoe
print()

for v in d.values():        # Iterate over values
    print(v, end= " ")      # 44 39 40 41
print()

for k,v in d.items():       # Iterate over  key-value pairs
    print(k, v, end= " ") # Fred 44 Ella 39 Owen 40 Zoe 41
print()
```

- Methods `d.keys()` and `d.values()` give access to keys and values
- Method `d.items()` gives access to key-value pairs
- Iteration is non-ordered $\Rightarrow$ they can come out in any order

Use dictionary for counting

Problem: Generate 1000 random integers in range 1-10 and print how many of each number that is generated.

```
import random as rnd

count = {}      # An empty dictionary
for i in range(1000):
    r = rnd.randint(1,10)
    if r not in count:  # If r not in dictionary .
        count[r] = 0    # ... add with count zero
    count[r] += 1       # Update count for key r

for k,v in count.items():
    print(f"{k}\t{v}")  # tab separated k,v print
```

Output

```
5       95
10      104
6       101
8       110
3       96
1       88
4       100
7       108
2       94
9       104
```

Dictionaries are excellent for counting how many x,y,z, ... we have found. Just remember to initialize a new key-value pair (x,0) the first time you see a new x.

Grouping elements using dictionaries

Problem: Generate 20 random integers in range 1-100 and group them as *odd* or *even*.

```
import random as rnd

# Initialize groups with two empty lists
groups = {"odd":[], "even":[]}

for i in range(20):
    r = rnd.randint(1,100)
    if r % 2 == 0:
        groups["even"] += [r]    # Add r to even list
    else:
        groups["odd"] += [r]     # Add r to odd list

for k,v in groups.items():
    print(f"{k}\t{v}")      # Tab separated print
```

Output

```
odd     [35, 27, 41, 1, 49, 67, 13, 63, 97, 29, 79]
even    [92, 54, 78, 10, 8, 72, 42, 100, 58]
```

`groups = {"odd":[], "even":[]}` ⇒ Two pairs where the values are empty lists.

Dictionary Comprehensions

List Comprehensions ⇒ a short (and fast) way to construct new lists
Dictionary Comprehensions ⇒ a short (and fast) way to construct new dictionaries

```
# List comprehension
lst = [n**3 for n in range(2,11,2)]     # n = 2,4,6,8,10
print(lst)

# Dictionary comprehension
dct = {n:n**3 for n in range(2,11,2)}
print(dct)
```

Output

```
[8, 64, 216, 512, 1000]
{2: 8, 4: 64, 6: 216, 8: 512, 10: 1000}
```

Hence, we generate pairs (`n:n**3`) rather than single elements (`n**3`) when we use dictionary comprehensions. The result is a new dictionary.

Data Structure Summary

- Lists, tuples, sets, and dictionaries are all data structures
- Each of them has their own set of properties
- Lists are sequential, mutable and very flexible
- Tuples (a fast and light version of lists) are sequential and immutable
- Sets are non-ordered, non-sequential, has no duplicate elements, and provides a fast look up (compared to lists and tuples)
- Dictionaries are a non-ordered collection of key-value pairs designed to provide a fast look up of values for a certain key.

- Consider the data structures as tools in a toolbox. Learn their properties and how to use them.
- List are most frequently used, shortly followed by dictionaries.
  They are both great!

The `zip` function

```
teachers = ("Jonas", "Ola", "Tobias", "Morgan", "Fredrik")
ages = (56, 55, 43, 45)    # No age for Fredrik!

lst = list( zip(teachers,ages) )  # Create list of tuples
print(lst)  # [('Jonas', 56), ('Ola', 55), ('Tobias', 43), ('Morgan', 45)]

lst = [1,2,3,4]
tpl = tuple( zip(lst,ages) )       # Zip list and tuple ==> tuple of tuples
print(tpl)                         # ((1, 56), (2, 55), (3, 43), (4, 45))

s = "Hello"
lst = list( zip(lst,s) )         # Zip list and string ==> tuple
print(lst)                       # [(1, 'H'), (2, 'e'), (3, 'l'), (4, 'l')]
```

- Sequential data like strings, lists, and tuples can be zipped
- `zip(a,b)` $\Rightarrow$ form a sequence of tuple pairs, one element from each of a and b.
- The shorter sequence decides the length of the result
- The zip result can be a list (using `list()`) or a tuple (using `tuple()`)

The `join` method for strings

```
teachers = ("Jonas", "Ola", "Tobias", "Morgan", "Fredrik")
print( " x ".join(teachers) )

lst = [1,2,3,4]
as_strings = [str(n) for n in lst]
print( " --> ".join(as_strings))

s = "Hello"
print( ",".join(s))
```

Output

```
Jonas x Ola x Tobias x Morgan x Fredrik
1 --> 2 --> 3 --> 4
H,e,l,l,o
```

- The join() string method returns a string by joining all the elements of a string sequence, separated by a string separator.
- Syntax: `sep_string.join( string_sequence )` where `string_sequence` is a sequence (list, tuple or string) containing strings, and `sep_string` is the string that will separate the elements in `string_sequence` once they been joined.

The keywords `None` and `pass`

```
def is_positive(n):
    if n == 0:
        return    # Will return None
    else:
        return n > 0

def tricky_function():      # Function with no return value
    # Not yet implemented
    pass                    # Compilation error if removed

# Program starts
print( is_positive(0) )     # Output: None

p = tricky_function()
print(p)                    # Output: None
```

- The keyword **pass** is a null (empty) statement. Not ignored by the interpreter but nothing will happens. The pass statement is useful when you don't write the implementation of a function but you want to implement it in the future.
- The **None** keyword is used to define a null value, or no value at all. Operations on None will give an error.

Assignment 3 information

Due to technical problems with Gitlab and us realizing that that the deadline is much too tough, things have changed.

- A new deadline for Assignment 3 is Wednesday, October 7, at 23.55.
- We use the Moodle submission system to submit Assignment. It is easy, just zip the folder `YourUsername_assign3` into a zip file and upload it using the Assignment 3 submission in Moodle.
- Assignment 3 will be graded using the ECTS scale A-F (where F is Fail) and we require you to handle all exercises marked as "If time permits" to get the two highest grades, A or B, and you need to handle all mandatory exercises to avoid an F.