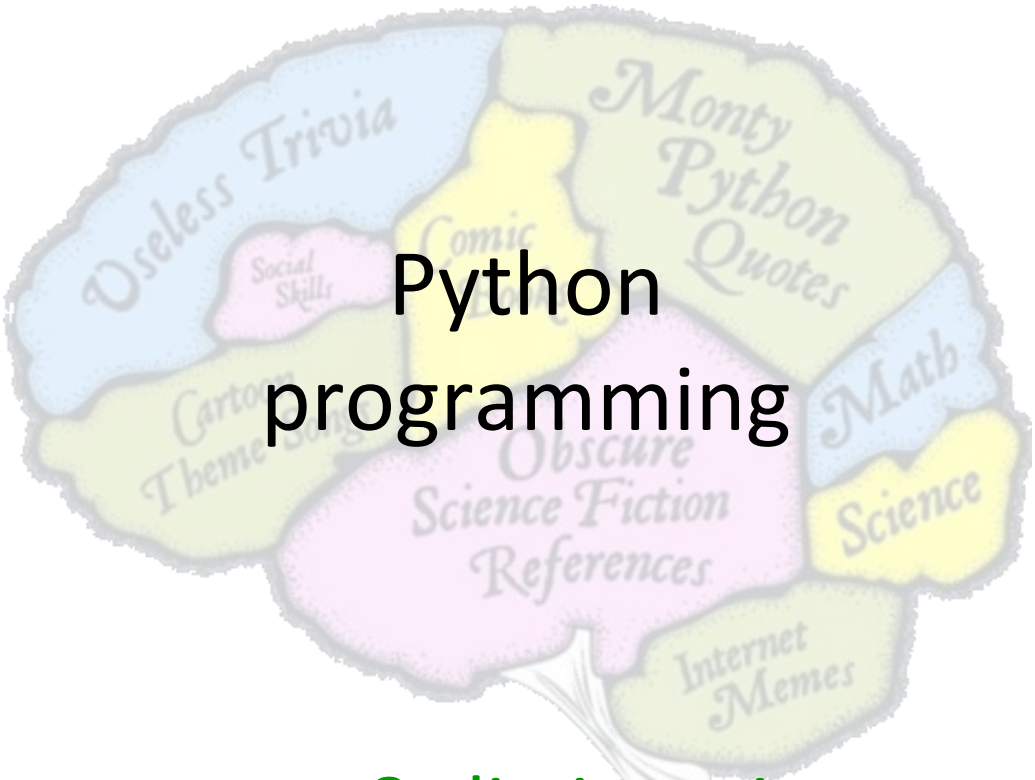
Ghent University
Faculty of Sciences



Python programming

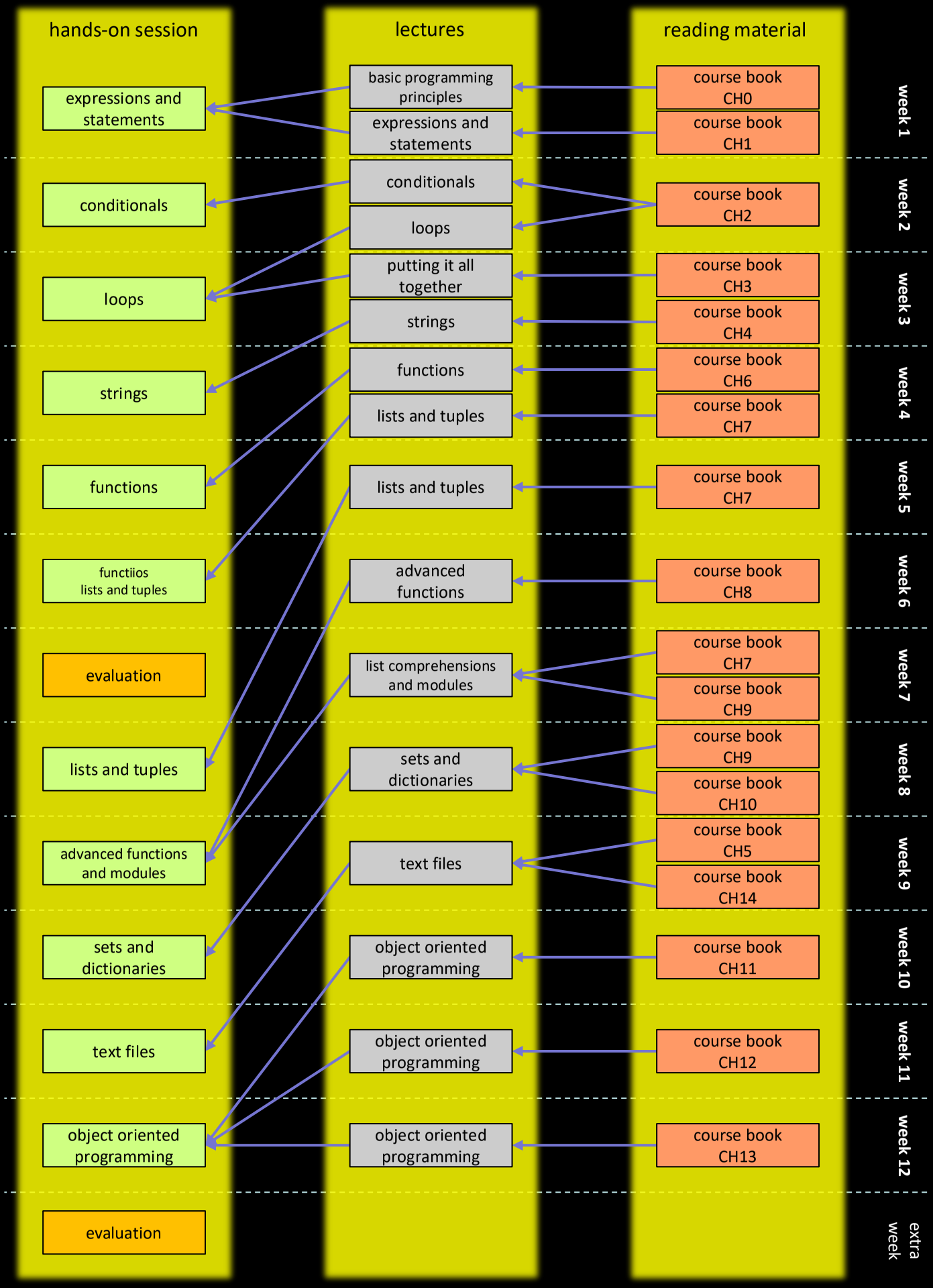
sets & dictionaries

Prof. Dr. Peter Dawyndt

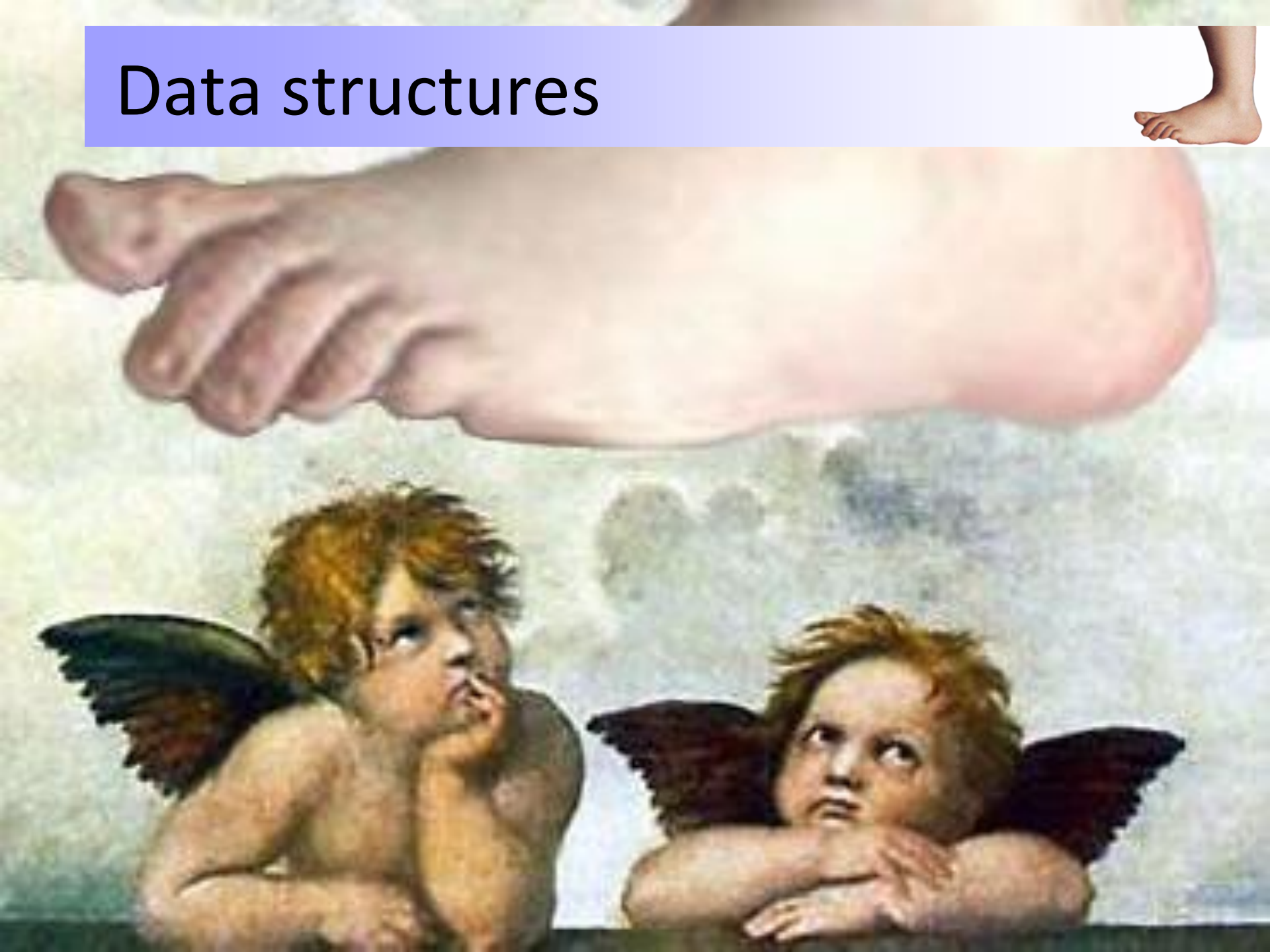
peter.dawyndt@ugent.be

@dawyndt

UNIVERSITEIT GENT

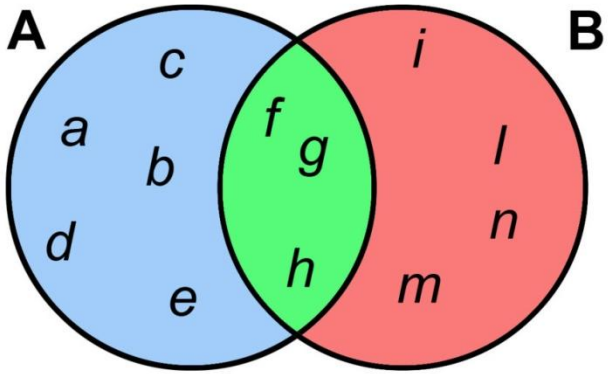

hands-on session
lectures
reading material
week 1
week 2
week 3
week 4
week 5
week 6
week 7
week 8
week 9
week 10
week 11
week 12
extra week
expressions and statements
conditionals
loops
strings
functions
functiios lists and tuples
evaluation
lists and tuples
advanced functions and modules
sets and dictionaries
text files
object oriented programming
evaluation
basic programming principles
expressions and statements
conditionals
loops
putting it all together
strings
functions
lists and tuples
lists and tuples
advanced functions
list comprehensions and modules
sets and dictionaries
text files
object oriented programming
object oriented programming
object oriented programming
course book CH0
course book CH1
course book CH2
course book CH3
course book CH4
course book CH6
course book CH7
course book CH7
course book CH8
course book CH7
course book CH9
course book CH9
course book CH10
course book CH5
course book CH14
course book CH11
course book CH12
course book CH13

Data structures



Data structures

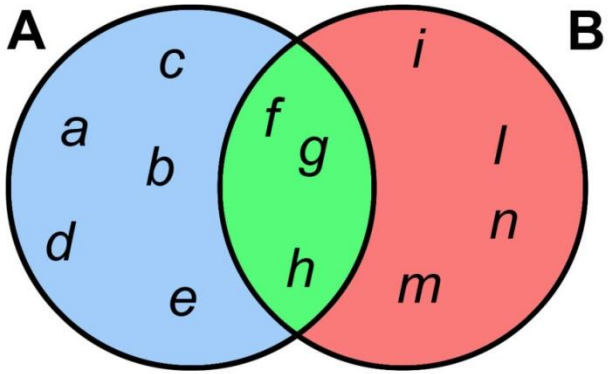
- the world is *not* only made of lists
  - just because it's the first data structure we have discussed, doesn't make it the right one for every task
  - this lecture introduces two other fundamental data structures
    - they will allow you to create programs that are simpler and more efficient
  - we will also discuss what "efficient" really means
- first new data structure: sets



UNIVERSITEIT GENT

Sets

- **set**: unordered collection of distinct objects
  - unordered: objects are looked up by value, rather than location
  - distinct: any object appears at most once
- fundamental concept in mathematics
  - an afterthought in most programming languages
  - but essential in problem solving



UNIVERSITEIT GENT

Sets

- **`set`** data type is built into Python
  - create a new set by calling **`set()`** or **`{e1, e2, …}`**
  - insert and remove objects, test for membership, …
  - **`set(i)`** converts *iterable object* **`i`** into a set

```
>>> vowels = set()
>>> for character in 'aieoeiaoaaeieou':
...     vowels.add(character)
...
>>> vowels
{'a', 'i', 'e', 'u', 'o'}
>>> set('unconsciousness')
{'c', 'e', 'i', 'o', 'n', 's', 'u'}
>>> {x // 2 for x in range(10)}
{0, 1, 2, 3, 4}
```

UNIVERSITEIT GENT

Set operations

- like other objects, sets have methods
  - following methods change set *in place* and return value **None**

```
>>> lows = {0, 1, 2, 3, 4}
>>> lows.remove(0)              # remove an element
>>> lows
{1, 2, 3, 4}
>>> lows.add(3)                 # add an element
>>> lows.add(9)
>>> lows
{1, 2, 3, 4, 9}
>>> lows.clear()                # remove all elements
>>> lows
set()
```

UNIVERSITEIT GENT

Set operations

- like other objects, sets have methods
  - following methods return new set

```
>>> tens = set(range(10))
>>> lows = {0, 1, 2, 3, 4}
>>> odds = {1, 3, 5, 7, 9}
>>> lows.difference(odds)                # difference
{0, 2, 4}
>>> lows
{0, 1, 2, 3, 4}
>>> odds.difference(lows)                # asymmetric
{9, 5, 7}
>>> lows.symmetric_difference(odds) # symmetric
{0, 2, 4, 5, 7, 9}
```

UNIVERSITEIT GENT

Set operations

- like other objects, sets have methods
  - following methods return new set
  - can also be expressed using operators

```
>>> tens = set(range(10))
>>> lows = {0, 1, 2, 3, 4}
>>> odds = {1, 3, 5, 7, 9}
>>> lows - odds                          # difference
{0, 2, 4}
>>> lows
{0, 1, 2, 3, 4}
>>> odds - lows                          # asymmetric
{9, 5, 7}
>>> lows ^ odds                          # symmetric
{0, 2, 4, 5, 7, 9}
```

UNIVERSITEIT GENT

Set operations

- like other objects, sets have methods
  - following methods return new set

```
>>> tens = set(range(10))
>>> lows = {0, 1, 2, 3, 4}
>>> odds = {1, 3, 5, 7, 9}
>>> lows.intersection(odds)          # intersection
{1, 3}
>>> lows.union(odds)                 # union
{0, 1, 2, 3, 4, 5, 7, 9}
>>> lows.issubset(tens)              # subset
True
>>> lows.issuperset(odds)           # superset
False
```

UNIVERSITEIT GENT

Set operations

- like other objects, sets have methods
  - following methods return new set
  - can also be expressed using operators

```
>>> tens = set(range(10))
>>> lows = {0, 1, 2, 3, 4}
>>> odds = {1, 3, 5, 7, 9}
>>> lows & odds                          # intersection
{1, 3}
>>> lows | odds                          # union
{0, 1, 2, 3, 4, 5, 7, 9}
>>> lows <= tens                         # subset
True
>>> lows >= odds                         # superset
False
```

UNIVERSITEIT GENT

Iterating sets

- sets are *iterable objects*

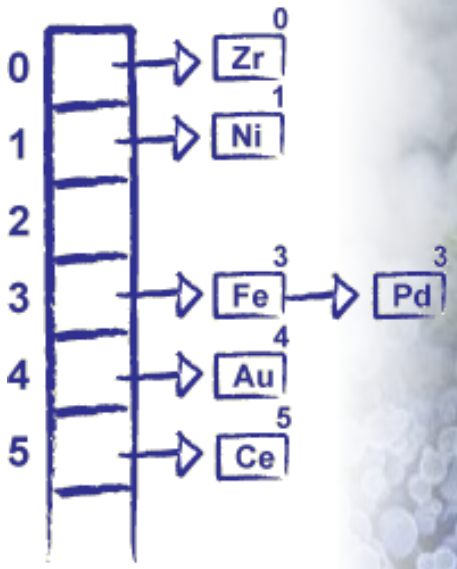
```
>>> monty_python = ['Graham Chapman', 'Eric Idle',
...                 'Terry Gilliam', 'Terry Jones',
...                 'John Cleese', 'Michael Palin']
>>> firstnames = [v for v, f in
...     [name.split() for name in monty_python]]
>>> firstnames
['Graham','Eric','Terry','Terry','John','Michael']
>>> unique_names = set(voornamen)
>>> for name in unique_names:
...     print(name, end=' ')
...
Michael John Graham Eric Terry
```

UNIVERSITEIT GENT

How set values are stored

- implementation goal
  - make lookup as quick as possible
  - without making insertion and removal expensive

- possible solution: *hash table*
  - compute a hash value for every object being inserted
  - store object at that location in list
  - collisions → objects grouped in sublist
  - good hash function → collisions will be rare

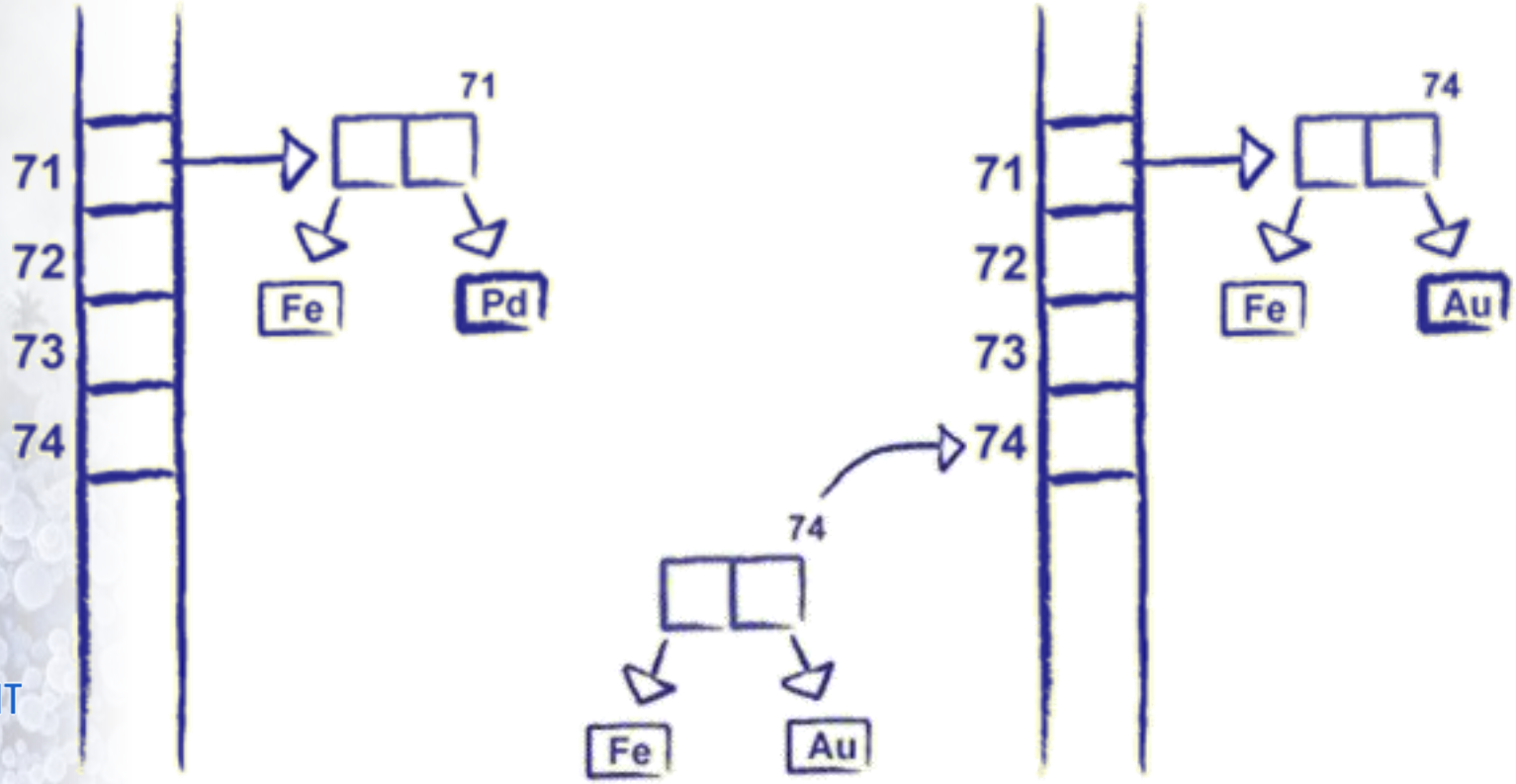


  - result: looking up an object takes *constant time*, regardless of how many objects are being stored

UNIVERSITEIT GENT

Immutability

- this only works if the hash value of an object never changes after it is inserted
  - otherwise the object will be stored in the wrong place



UNIVERSITEIT GENT

Immutability

- this only works if the hash value of an object never changes after it is inserted
  - otherwise the object will be stored in the wrong place
  - Python therefore only allows sets to contain immutable objects
    - booleans, numbers, strings, tuples, …
    - … but **not** lists

```
>>> monty_python = set()
>>> monty_python.add('Graham Chapman')
>>> monty_python
{'Graham Chapman'}
>>> monty_python.add(('Eric', 'Idle'))
>>> monty_python
{'Graham Chapman', ('Eric', 'Idle')}
>>> monty_python.add(['Terry', 'Gilliam'])
TypeError: unhashable type: 'list'
```

UNIVERSITEIT GENT

Immutability

- this only works if the hash value of an object never changes after it is inserted
  - otherwise the object will be stored in the wrong place
  - Python therefore only allows sets to contain immutable objects
    - this is one of the reasons tuples were invented
      - allows hashing of multi-part objects

```
>>> monty_python = set()
>>> monty_python.add('Graham Chapman')
>>> monty_python
{'Graham Chapman'}
>>> monty_python.add(('Eric', 'Idle'))
>>> monty_python
{'Graham Chapman', ('Eric', 'Idle')}
>>> monty_python.add(['Terry', 'Gilliam'])
TypeError: unhashable type: 'list'
```

UNIVERSITEIT GENT

Frozen sets

- what about sets of sets?
  - sets themselves have to be mutable, so that values can be inserted and removed
- Python also provides "frozen" sets
  - no changes allowed after creation
  - they're almost always initialized from a collection of some kind

```
>>> monty_python = set()
>>> terries = frozenset(['Terry Jones', 'Terry Gilliam'])
>>> monty_python.add(terries)
>>> monty_python
set([frozenset(['Terry Gilliam', 'Terry Jones'])])
>>> terries.add('Michael Palin')
AttributeError: 'frozenset' object has no attribute 'add'
```

UNIVERSITEIT GENT

A note on language design

- many languages (but not Python)
  allow mutable elements in sets
  - users are trusted no to modify them after insertion
  - rich source of hard-to-find-bugs
- some languages (but not Python)
  keep track of all sets an object is in
  - "move" them when their values change
  - makes all programs slow for the benefit of a few
- every software system
  contains tradeoffs like this

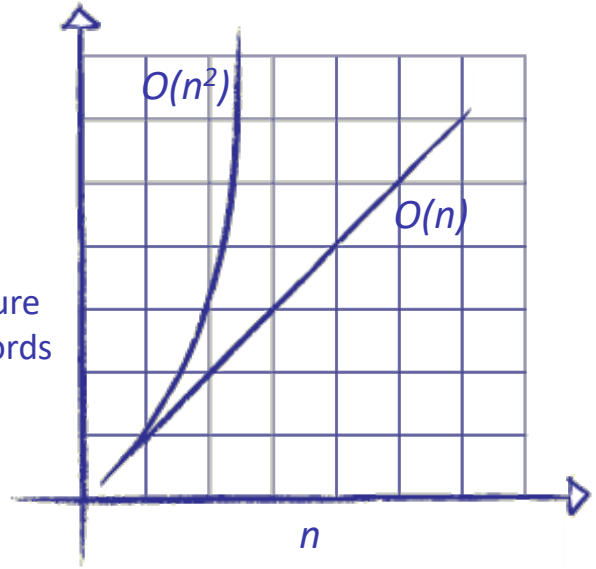
UNIVERSITEIT GENT

Efficiency

- so is using sets worthwhile?
  - construct data structure with *n* unique words

```
if word in words
```

  - `type(words) == list`
    - each check requires *n*/2 comparisons on average
    - building up those *n* unique words thus requires *n*(*n*+1)/4 comparisons
  - `type(words) == set`
    - build set with *n* unique words only takes *n* comparisons
    - difference is dramatic



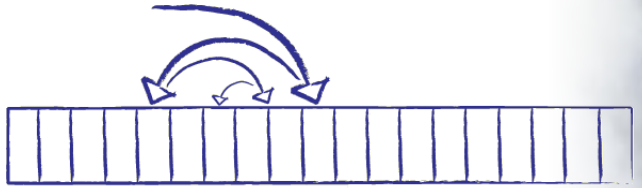
build data structure with *n* unique words

UNIVERSITEIT GENT

Efficiency

- so is using sets worthwhile?
  - construct data structure with *n* unique words

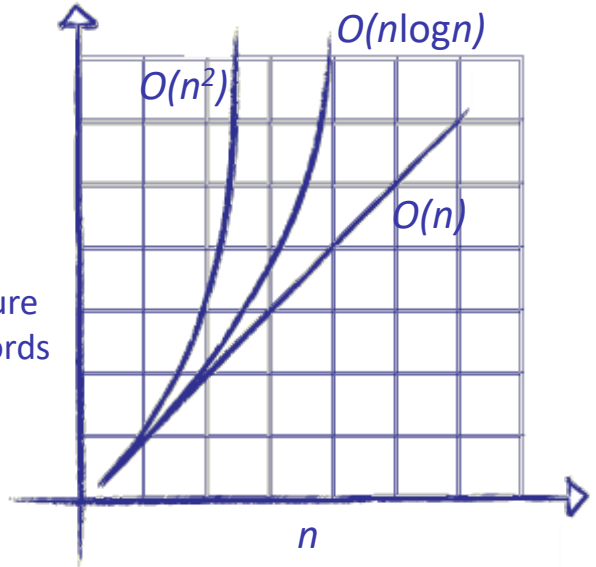
```
if word in words
```



binary search

  - `type(words) == list`
    - can get better performance if we keep list sorted
    - *k* checks is enough to find an object in a list of length 2^k
      - list containing *n* objects can be searched in $\log_2 n$ steps
      - building list of *n* unique words takes $n\log_2 n$ comparisons

build data structure with *n* unique words



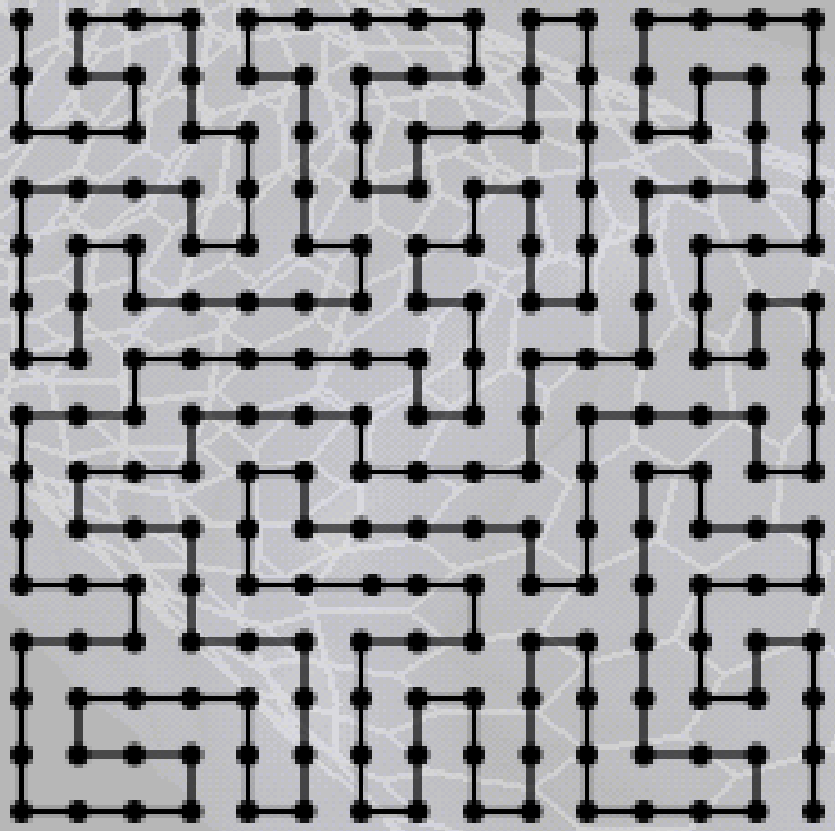
UNIVERSITEIT GENT

Algorithmic complexity

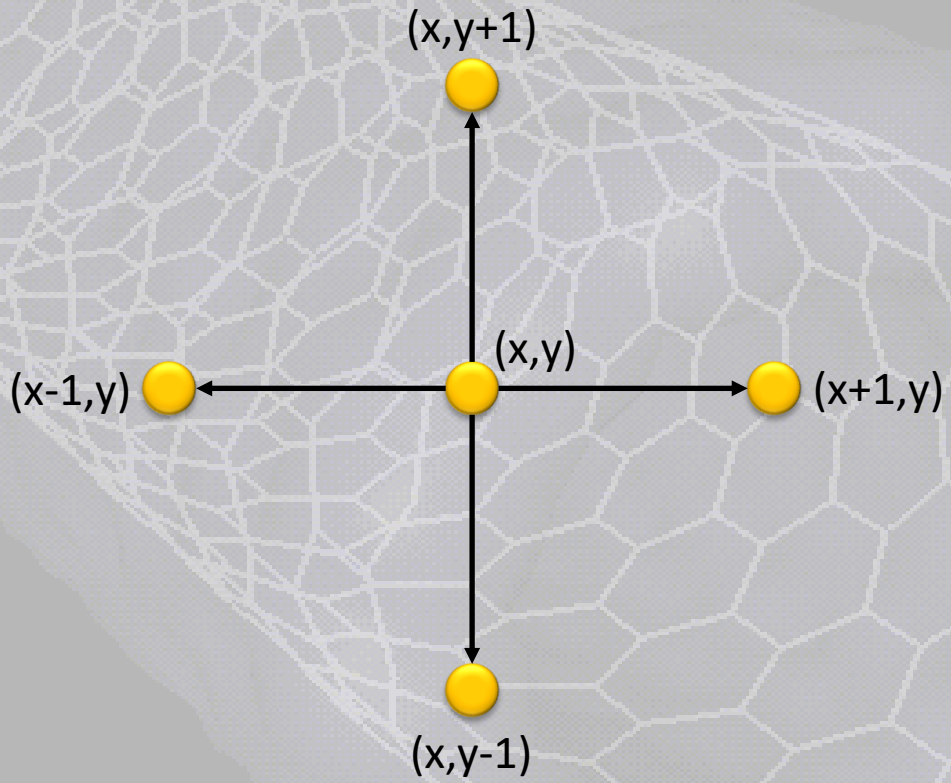
- algorithmic complexity
  - relationship between problem size and running time
  - usually described in terms of upper bounds
  - if $f(x) < kg(x)$ for large x and some constant k, then $f(x)$ is $O(g(x))$
- for example
  - $O(1)$ something that takes the same time regardless of the data size
  - $O(\log n)$ time grows as the logarithm of the data size
  - $O(n)$ time grows proportional to the number of objects
- storing unique words in a list is $O(n^2)$
  - $n(n+1)/4 = n^2/4 + n/4 \sim n^2$ as n grows large

UNIVERSITEIT GENT

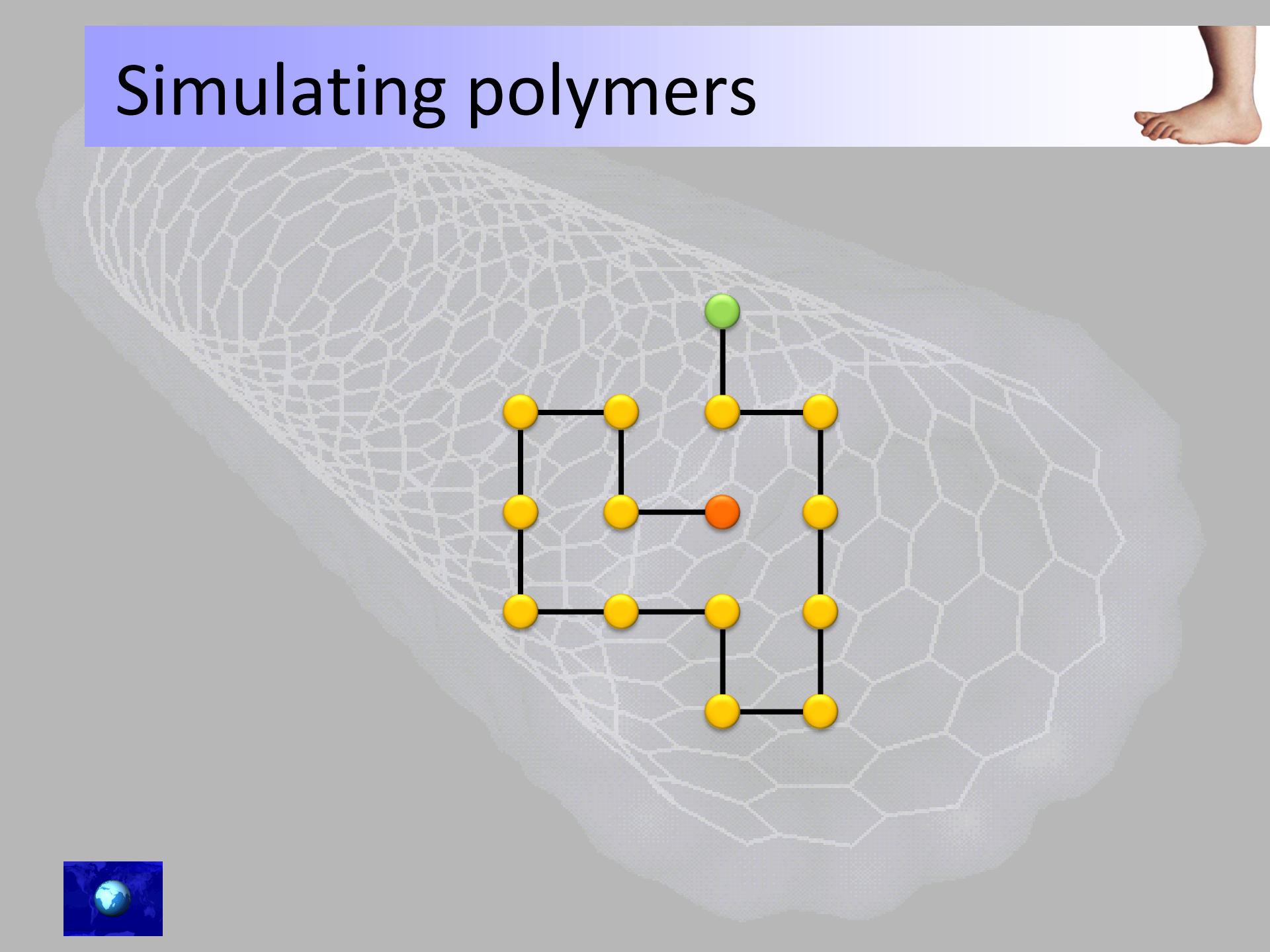
Simulating polymers



Simulating polymers

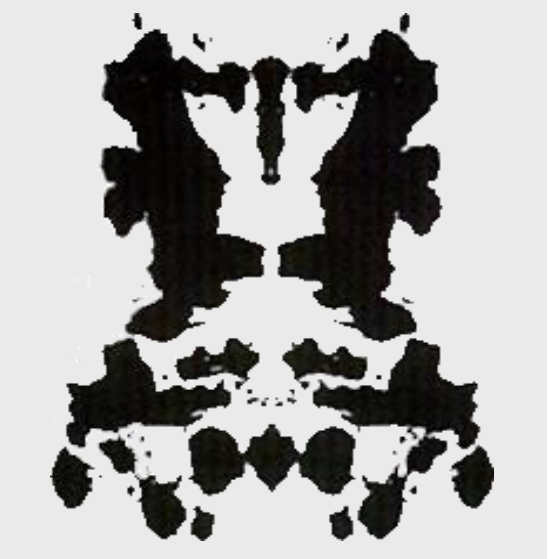


Simulating polymers



Dictionaries

Human associations



UNIVERSITEIT GENT

Human associations

APPLE

What are you associating this word with ?

UNIVERSITEIT GENT

Did you think about …

delicious

fruit

GREEN

UNIVERSITEIT GENT

… or rather about?

UNIVERSITEIT GENT

Logos after financial crisis

DOWNJONES

NOKIA
Disconnecting People

PLEASE BUY

2M

Will work for food

DEFAULT

citigroup

LG
Life's tough

CRISISLER

DELL

YAHOO?

BAD YEAR

UNIVERSITEIT GENT

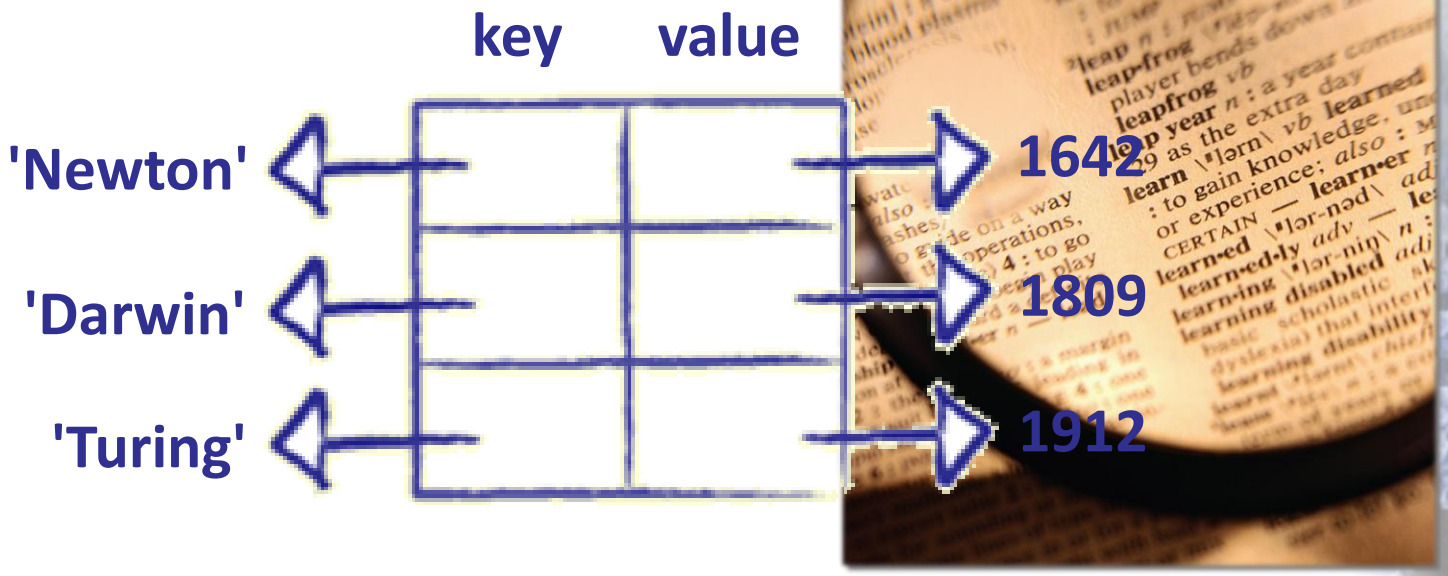
FIASCO SYSTEMS

Fail

Ferrari

Dictionaries

- **motivation**: store extra data with each element of a set
- **dictionary**: unordered mutable collection
  - not a sequence but a *mapping*
  - associates one *value* with each of its *keys*
  - synonyms: map, hash, associative array
  - often visualized as two-column table



UNIVERSITEIT GENT

Creating and indexing

- create a dictionary by pytting key/value pairs inside **{** and **}**
  - empty dictionaries are written as **{}**
- look up value associated with a key using **[** and **]**
  - only existing keys can be accessed
  - just as you can't index elements of a list that aren't there

```
>>> birthday = {'Newton':1642, 'Darwin':1809}
>>> f"Darwin's birthday: {birthday['Darwin']}"
Darwin's birthday: 1809
>>> f"Newton's birthday: {birthday['Newton']}"
Newton's birthday: 1642
>>> f"Turing's birthday: {birthday['Turing']}"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Turing'
```

UNIVERSITEIT GENT

Updating

- associating an object to a key
  - key does not exist: create a new key/value pair
  - key does exist: overwrite value associated with key
- remove entry: **`del d[`*`key`*`]`**
  - only entries that are actually present can be removed

```
>>> birthday = {}
>>> birthday['Darwin'] = 1809
>>> birthday['Newton'] = 1942      # oops
>>> birthday['Newton'] = 1642
>>> birthday['Turing'] = 1912
>>> birthday
{'Turing': 1912, 'Darwin': 1809, 'Newton': 1642}
>>> del birthday['Turing']
>>> birthday
{'Darwin': 1809, 'Newton': 1642}
>>> del birthday['Faraday']
KeyError: 'Faraday'
```

UNIVERSITEIT GENT

Testing membership

- **`k in d`**
  - tests whether a key **`k`** occurs in dictionary **`d`**
  - once again, inconsistent with behaviour of lists, but useful

```
>>> birthday = {'Newton':1642, 'Darwin':1809}
>>> for name in ['Newton', 'Turing']:
...     if name in birthday:
...         print(name, birthday[name])
...     else:
...         print(f'Who is {name} ?')
...
Newton 1642
Who is Turing ?
```

UNIVERSITEIT GENT

Loops

- **`for s in d`**
  - iterates the keys of dictionary **`d`** (rather than its values)
  - different from lists, where **`for`** loops over the values, rather than the indices

```
>>> birthday = {
...       'Newton' : 1642,
...       'Darwin' : 1809,
...       'Turing' : 1912
... }
>>> for name in birthday:
...       print(name, birthday[name])
...
Turing 1912
Newton 1642
Darwin 1809
```

UNIVERSITEIT GENT

Dictionary methods

- yes, dictionaries are objects too …

```
>>> birthday = {'Newton':1642,'Darwin':1809,'Turing':1912}
```

method	purpose	example	result
`clear`	empty the dictionary	`d.clear()`	returns `None`, but `d` is now empty
`get`	return the value associated with a key, or a default value if the key is not present	`d.get('x', 99)`	returns `d['x']` if `'x' in d`, or `99` if it is not
`keys`	return the dictionary's keys as a list; entries are guaranteed to be unique	`birthday.keys()`	`['Turing', 'Newton', 'Darwin']`
`values`	return the dictionary's values as a list; entries may or may not be unique	`birthday.values()`	`[1912, 1642, 1809]`
`items`	return a list of key/value pairs as an iterable of tuples	`birthday.items()`	`[('Turing', 1912), ('Newton', 1642), ('Darwin', 1809)]`
`update`	copy keys and values from one dictionary into another	`d2 = {}` `d2.update(d)`	

Dictionary methods

```
>>> birthday = {
...     'Newton' : 1642,
...     'Darwin' : 1809,
...     'Turing' : 1912
... }
>>> f'keys: {list(birthday.keys())}')
keys: ['Turing', 'Newton', 'Darwin']
>>> f'values: {list(birthday.values())}')
values: [1912, 1642, 1809]
>>> f'items: {list(birthday.items())}'
items: [('Turing', 1912), ('Newton', 1642), ('Darwin', 1809)]
>>> f'get: {birthday.get('Curie', 1867)}'
get: 1867
>>> temp = {'Curie':1867, 'Hopper':1906, 'Franklin':1920}
>>> birthday.update(temp)
>>> birthday
{'Curie': 1867, 'Darwin': 1809, 'Franklin': 1920,
 'Turing': 1912, 'Newton': 1642, 'Hopper': 1906}
>>> temp.clear()
>>> temp
{}
```

UNIVERSITEIT GENT

Cryptograms

C	h		l	d							o	w	
Q	m	v	r	b	w	l	f		x	w	k	d	

f	a	s			r			k					
i	o	p	z	l	w		v	f		z	m	l	

	p		i		g	t			k	e	.		
p	c	w	v	f	x	z	v	y	l	.			

Histogram construction

observations1.py

```
# birds that need to be counted
birds = ['tern', 'goose', 'goose', 'hawk',
         'tern', 'goose', 'tern']

# build a dictionary of frequencies
observations = {}
for bird in birds:
    if bird in observations:
        # already seen, so increment by 1
        observations[bird] += 1
    else:
        # never seen before, so add to dictionary
        observations[bird] = 1

# display dictionary
print(observations)
```

```
{'hawk': 1, 'goose': 3, 'tern': 3}
```

UNIVERSITEIT GENT

A slight simplification

observations2.py

```
# birds that need to be counted
birds = ['tern', 'goose', 'goose', 'hawk',
         'tern', 'goose', 'tern']

# build a dictionary of frequencies
observations = {}
for bird in birds:
    # dict.get: fetch bird observations
    # with default value 0
    observations[bird] = observations.get(bird, 0) + 1



# display dictionary
print(observations)
```

```
{'hawk': 1, 'goose': 3, 'tern': 3}
```

UNIVERSITEIT GENT

Imposing order

- a dictionary's keys are unordered (~ set elements)
  - remember, sets randomize (hash) their elements for fast lookup
  - so, to print counts in alphabetic order
    - get the list of keys
    - sort that list
    - loop over it

```
# display dictionary in alphabetic order
birds = list(observations.keys())
birds.sort()
for bird in birds:
    print(bird, observations[bird])
```

```
goose 3
hawk 1
tern 3
```

UNIVERSITEIT GENT

Imposing order

- a dictionary's keys are unordered (~ set elements)
  - remember, sets randomize (hash) their elements for fast lookup
  - so, to print counts in alphabetic order
    - get the list of keys
    - sort that list
    - loop over it

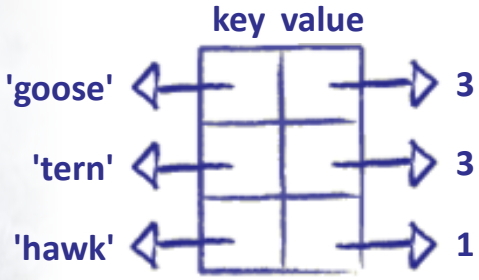
```
# display dictionary in alphabetic order


for bird in sorted(observations):
    print(bird, observations[bird])
```

```
goose 3
hawk 1
tern 3
```

UNIVERSITEIT GENT

Inverting a dictionary

- but how to print in order of frequency ?
  - need to invert dictionary: swap keys and values

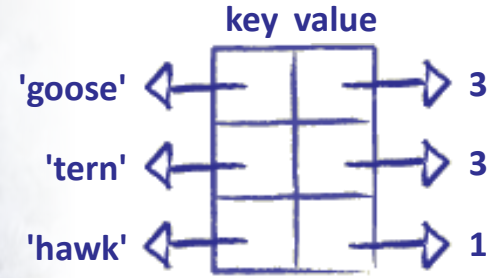


observations

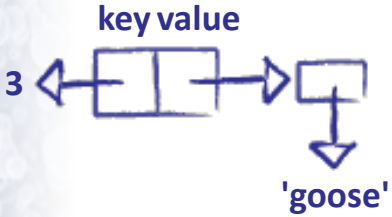
UNIVERSITEIT GENT

Inverting a dictionary

- but how to print in order of frequency ?
  - need to invert dictionary: swap keys and values
  - there might be collisions, since values aren't necessarily unique
    - what is the inverse of `{'a':1, 'b':1, 'c':1}` ?



observations

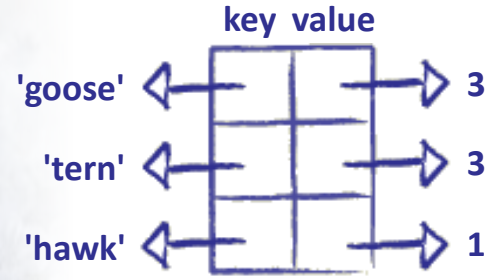


inverse

UNIVERSITEIT GENT

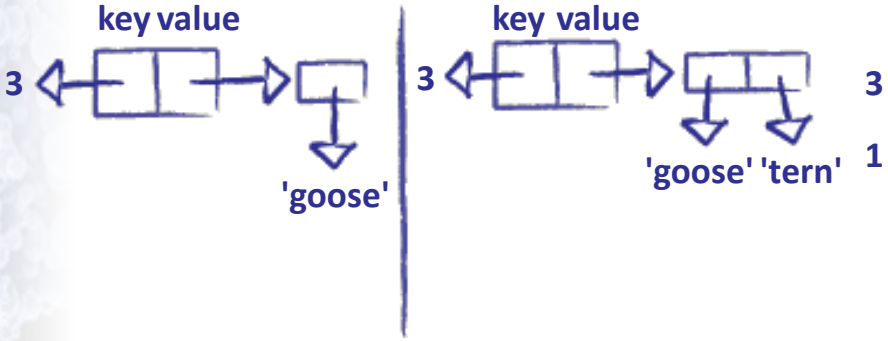
Inverting a dictionary

- but how to print in order of frequency ?
  - need to invert dictionary: swap keys and values
  - there might be collisions, since values aren't necessarily unique
    - what is the inverse of `{'a':1, 'b':1, 'c':1}` ?
  - solution: store a list of values instead of just a single value
    - use **`dict.get(key, [])`** instead of **`dict.get(key, 0)`**



observations

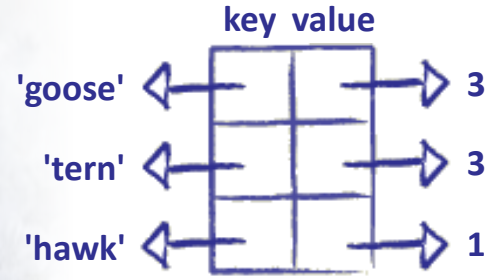
inverse



UNIVERSITEIT GENT

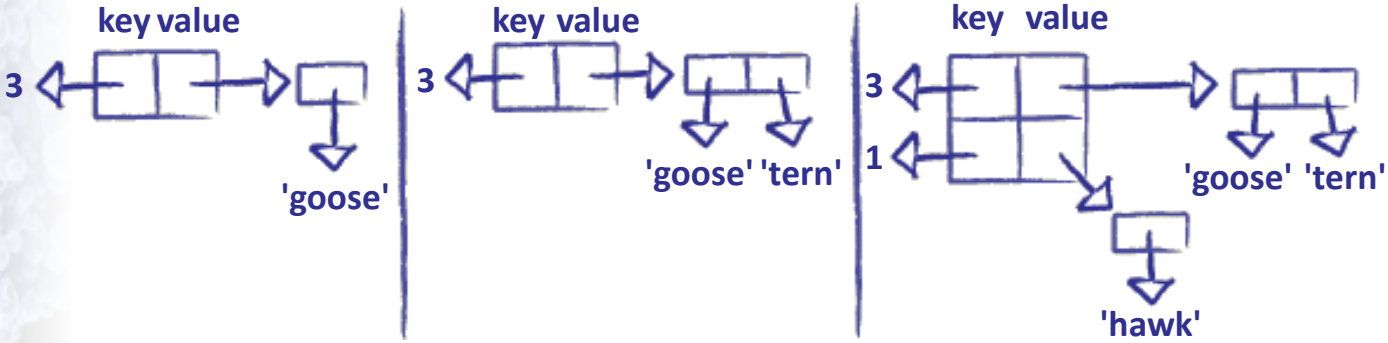
Inverting a dictionary

- but how to print in order of frequency ?
  - need to invert dictionary: swap keys and values
  - there might be collisions, since values aren't necessarily unique
    - what is the inverse of `{'a':1, 'b':1, 'c':1}` ?
  - solution: store a list of values instead of just a single value
    - use `dict.get(key, [])` instead of `dict.get(key, 0)`



observations

inverse



UNIVERSITEIT GENT

Inverting a dictionary

- but how to print in order of frequency ?
  - need to invert dictionary: swap keys and values

```
# invert dictionary
inverse = {}
for key, value in observations.items():
    birds = inverse.get(value, [])
    birds.append(key)
    inverse[value] = birds

# display dictionary in sorted order
for count in sorted(inverse):
    print(count, inverse[count])
```

```
1 ['hawk']
3 ['goose', 'tern']
```

UNIVERSITEIT GENT

Another way to do it

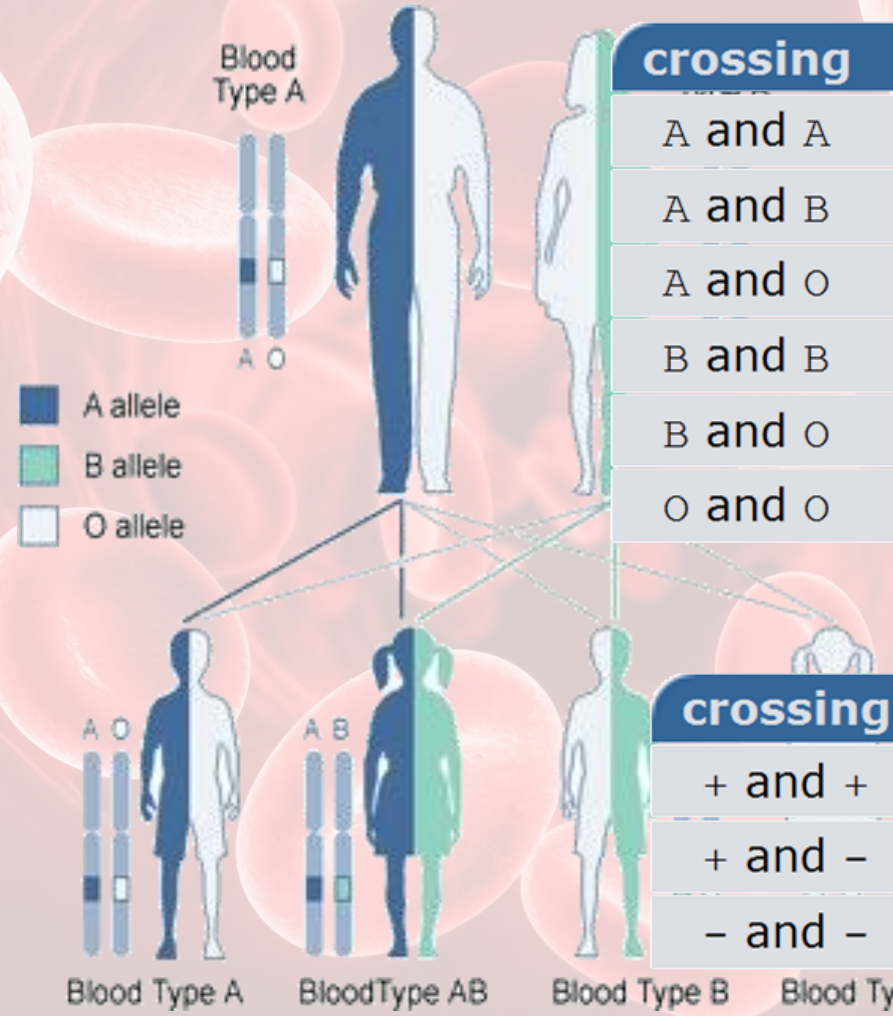
- but how to print in order of frequency ?
  - need to invert dictionary: swap keys and values

```
# invert dictionary
inverse = {}
for key, value in observations.items():
    if value not in inverse:
        inverse[value] = []
    inverse[value].append(key)

# display dictionary in sorted order
for count in sorted(inverse):
    print(count, inverse[count])
```

```
1 ['hawk']
3 ['goose', 'tern']
```

UNIVERSITEIT GENT

Blood types



crossing	ABO blood group
A and A	A
A and B	AB
A and O	A
B and B	B
B and O	B
O and O	O

crossing	Rhesus factor
+ and +	+
+ and –	+
– and –	–

Formatting strings with dicts

- complex string formatting can be hard to read
  - especially if one value needs to be used several times
  - use keyword arguments with string method **`format`**
    - use **`{key}`** inside format string to identify what has to be substituted at this position

```
birthday = {'Newton':1642, 'Darwin':1809, 'Turing':1912}
opmaak = '{name}: {year}'
for name, year in birthday.items():
    print(opmaak.format(name=name, year=year))
```

```
Turing: 1912
Newton: 1642
Darwin: 1809
```

UNIVERSITEIT GENT

Additional keyword arguments

- the ** before **keywords** means
  - put additional keyword arguments in dictionary **keywords**
- supports functions with variable number of arguments

```
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
```

```
def cheeseshop(cheese, **keywords):
    print(f'-- Do you have any {cheese} ?')
    print(f"-- I'm sorry, we're all out of {cheese}")
    print('-' * 40)
    for key in sorted(keywords):
        print(f'{key}: {keywords[key]}')

cheeseshop(
    'Limburger',
    client='John Cleese',
    shopkeeper='Michael Palin',
    skecth='Cheese Shop Sketch'
)
```

UNIVERSITEIT GENT

Additional positional arguments

- the * before **arguments** means
  - put additional positional arguments in tuple **arguments**
- there may be only one * per functie

```
def cheeseshop(cheese, *arguments, **keywords):
    print(f"-- Do you have any {cheese}?")
    print("I'm sorry, we're all out of {cheese}")
    for argument in arguments:
        print(argument)
    print('-' * 40)
    for key in sorted(keywords):
        print(f'{key}:{keywords[key]}')
```

```
cheeseshop(
    'Limburger',
    "It's very runny, sir.",
    "It's really very, VERY runny, sir.",
    client='John Cleese',
    shopkeeper='Michael Palin',
    skecth='Cheese Shop Sketch'
)
```

UNIVERSITEIT GENT

Homework (next lecture)

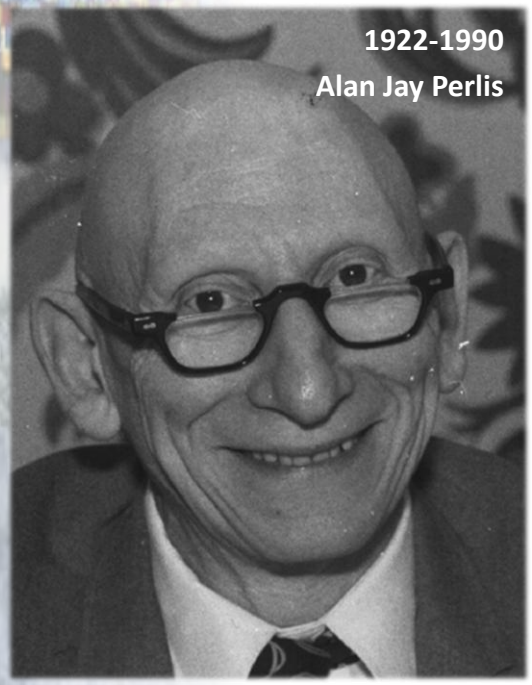
- course book
  - read chapter 5 (files and exceptions)
  - read chapter 14 (files and exceptions II)
- classroom exercises (series 09)
  - Alphabetic encoding
  - Cellophane
  - World Cup soccer

UNIVI
GENT

Questions or remarks?

UNIVERSITEIT GENT

The sky is the limit…



"If a listener nods his head when you're explaining your program, wake him up."

— Alan J. Perlis

UNIVERSITEIT GENT