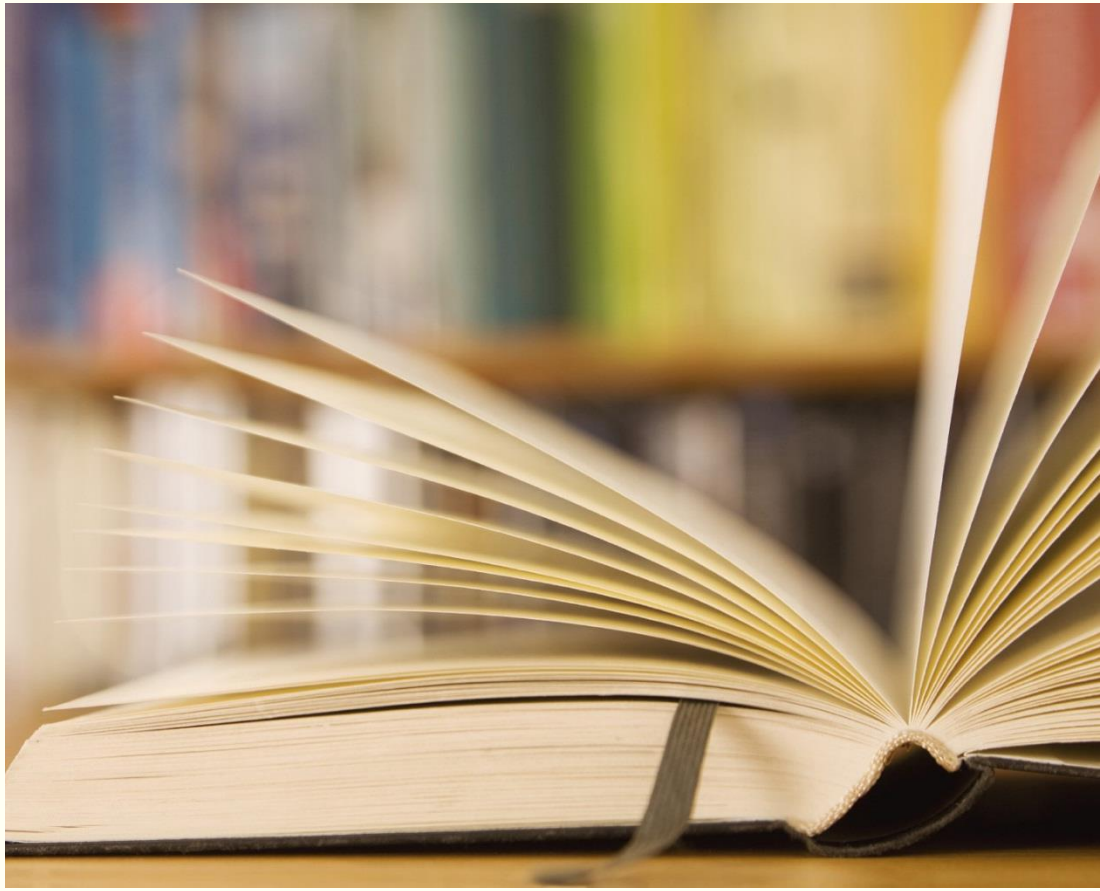
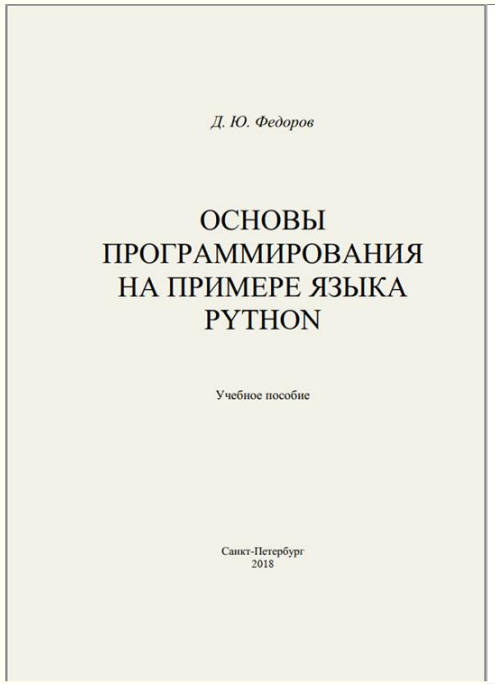
СТРУКТУРНЕ ПРОГРАМУВАННЯ МОВОЮ PYTHON

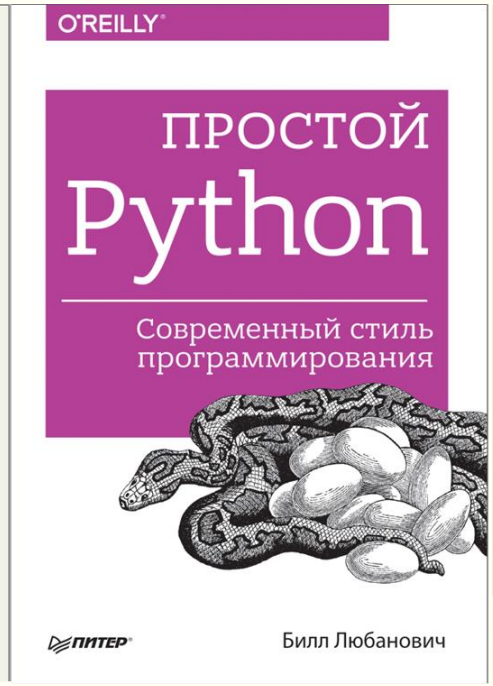
Лекція 07
Основи інформатики, програмування та алгоритмічні мови

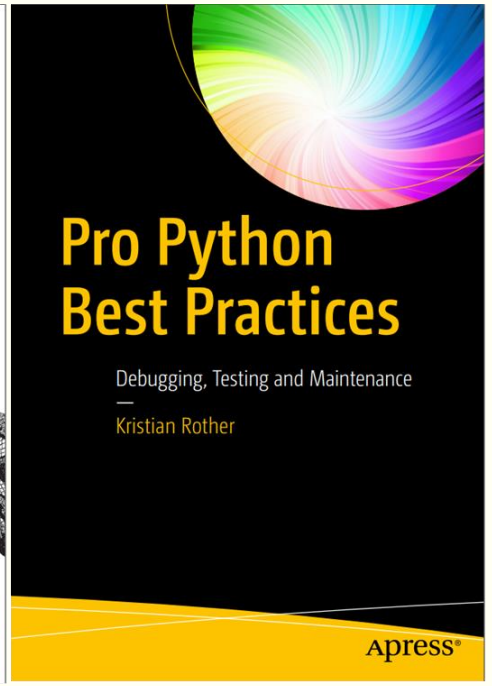


План лекції

- Вбудовані структури даних у мові Python.
- Організація Python-коду за допомогою функцій.
- Стратегії налагодження Python-коду
- Дослідницьке кодування та інструменти налагодження Python-коду.



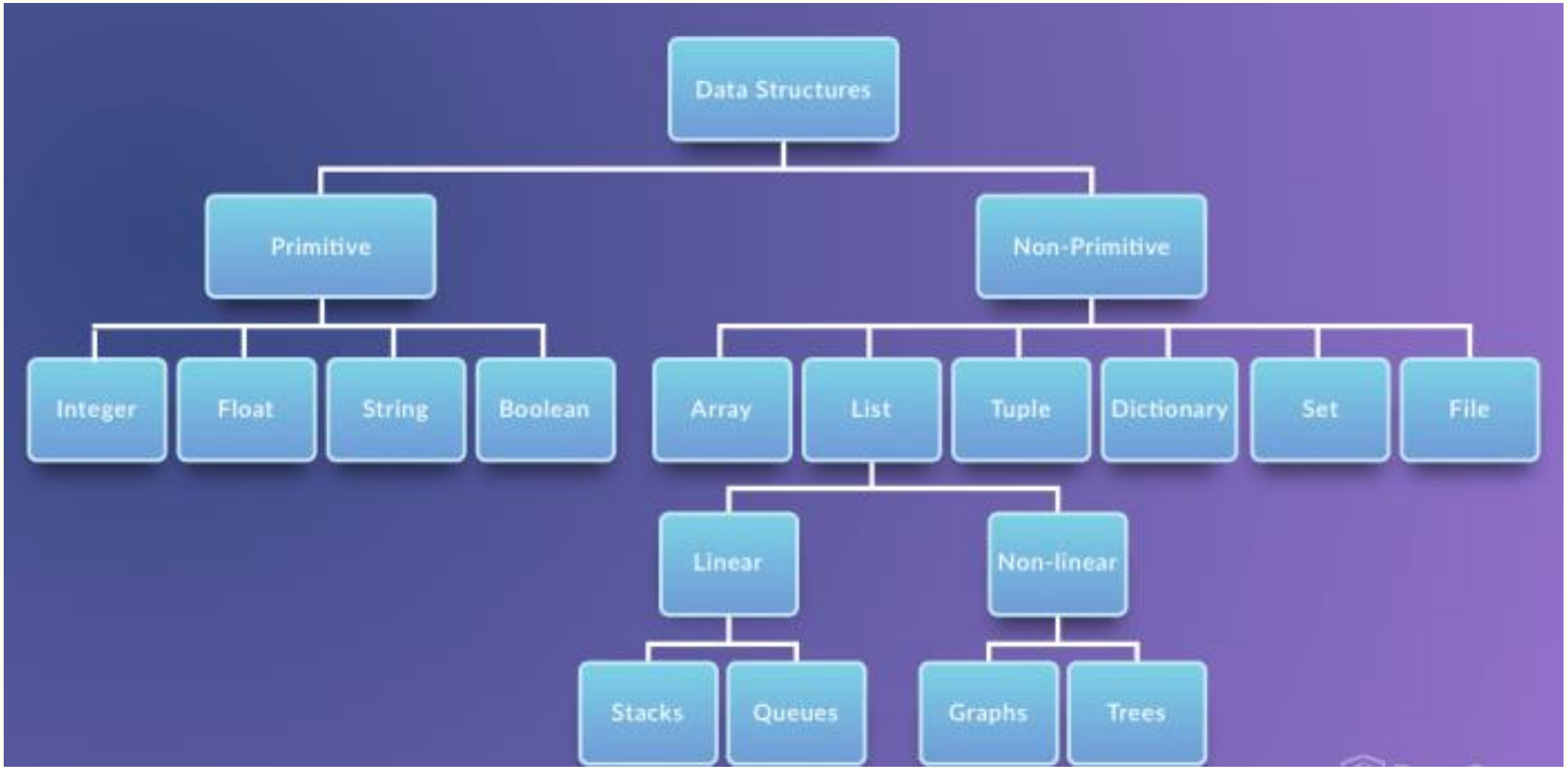




ВБУДОВАНІ СТРУКТУРИ ДАНИХ У МОВІ PYTHON

Питання 7.1

Структури даних у Python



Кортежі та іменовані кортежі

- ***Кортежі*** – це об’єкти, які можуть зберігати конкретну кількість інших об’єктів упорядковано.
  - Вони незмінювані (immutable), тому ми не можемо додати, видалити чи замінити об’єкти на льоту.
  - Основна перевага незмінюваності кортежів: можемо використовувати їх як ключі для словників.
- Кортежі зберігають ***лише дані***.
  - Якщо потрібна поведінка для управління кортежом, необхідно передати кортеж у функцію / метод, які виконають дію.
  - Основна мета кортежу – зібрати різні частини даних у межах одного контейнеру.
- Зазвичай кортежі оточені дужками, проте це не обов’язково.
  - Команди ідентичні:
  - stock = "FB", 75.00, 75.03, 74.90
  - stock2 = ("FB", 75.00, 75.03, 74.90)
- Якщо кортеж групується всередині певного іншого об’єкта (в основному, виклику функції), дужки потрібні.
  - Інакше буде неможливо інтерпретатору дізнатись, чи це кортеж, чи наступний параметр функції.

Приклад

- Функція приймає кортеж і дату, а повертає кортеж з дати та середньої ціни акції

```
import datetime
def middle(stock, date):
    symbol, current, high, low = stock
    return (((high + low) / 2), date)

mid_value, date = middle(("FB", 75.00, 75.03, 74.90),
        datetime.date(2014, 10, 31))
```

- Приклад також ілюструє розпаковку кортежу.
  - В останньому рядку повернений всередину функції кортеж розпаковується на 2 значення: mid_value та date.

Розпаковка – корисна риса мови Python

- Можна групувати змінні, щоб простіше зберігати та передавати їх, та розпакувати їх, коли потрібен доступ до них окремо.
  - Для отримання доступу до окремих значень можна використовувати той же синтаксис:

```
>>> stock = "FB", 75.00, 75.03, 74.90
>>> high = stock[2]
>>> high
75.03
```

  - Можна використовувати slice notation, щоб виділити частини кортежів:

```
>>> stock[1:3]
(75.00, 75.03)
```

Прямий доступ до членів кортежу корисний лише в окремих ситуаціях.

- Такі «магічні числа" – джерело помилок та плутанини при налагодженні коду.
- Принаймні, додайте коментарі з поясненням, звідки вони взяті.

Іменовані кортежі (named tuples)

- Потреба: згрупувати об’єкти з метою частого індивідуального доступу до них.
  - Зручно використовувати словник.
- Якщо в додаванні поведінки немає потреби, можна використати іменовані кортежі.
  - Добре підходять для поєднання read-only даних.
- Спочатку імпортуємо namedtuple та створюємо відповідний об’єкт.
  - Конструктор приймає ідентифікатор іменованого кортежу та пробільний рядок з атрибутами, що можуть бути в кортежі.
  - Утворений об’єкт можна інстанціювати, як звичайний екземпляр класу

```
from collections import namedtuple
Stock = namedtuple("Stock", "symbol current high low")
stock = Stock("FB", 75.00, high=75.03, low=74.90)
```

Іменовані кортежі (named tuples)

- З отриманим іменованим кортежом можна працювати, як зі звичайним кортежом, проте доступ до окремих атрибутів аналогічний об'єктам:

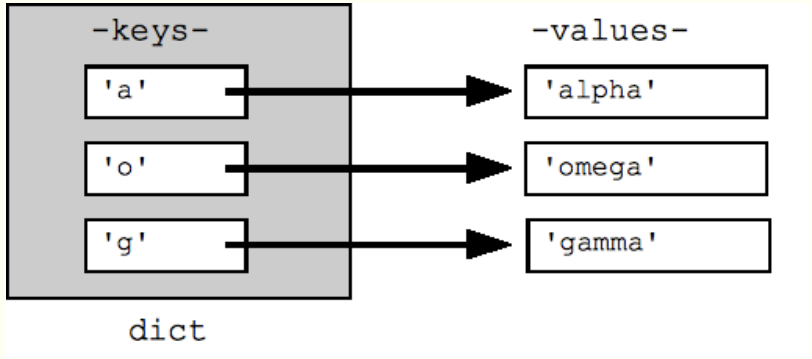
```
>>> stock.high
75.03
>>> symbol, current, high, low = stock
>>> current
75.00
```

- Як і кортежі та рядки, іменовані кортежі незмінювані (immutable).

```
>>> stock.current = 74.98
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Для змінюваних даних краще словники

- Відображають (map) об'єкти на інші об'єкти:
  - Словники особливо ефективні при пошуку значення за його ключем.
- Створюються за допомогою конструктора dict() або синтаксичного скорочення {}.
  - На практиці зазвичай використовують останню форму.
  - Всередині ключ від значення відділяється двокрапкою, а пари «ключ-значення» - комами.
  - stocks = {"GOOG": (613.30, 625.86, 610.50),
    "MSFT": (30.25, 30.70, 30.19)}
  - Якщо ключа у словнику немає,
  - отримаємо виключення:




```
>>> stocks["GOOG"]
(613.3, 625.86, 610.5)
>>> stocks["RIM"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'RIM'
```

Словники мають кілька пов'язаних з ними поведінок

- Метод get() приймає ключ та опційне значення за замовчуванням, якщо такого ключа немає:

```
>>> print(stocks.get("RIM"))
None
>>> stocks.get("RIM", "NOT FOUND")
'NOT FOUND'
```

- Для ще більшого контролю застосовують метод setdefault().
  - Якщо ключ є в словнику, метод працює як get().
  - Інакше не тільки поверне значення за замовчуванням, а й додасть його у словник:

```
>>> stocks.setdefault("GOOG", "INVALID")
(613.3, 625.86, 610.5)
>>> stocks.setdefault("BBRY", (10.50, 10.62, 10.39))
(10.50, 10.62, 10.39)
>>> stocks["BBRY"]
(10.50, 10.62, 10.39)
```

Словники мають кілька пов'язаних з ними поведінок

- Інші корисні методи: keys(), values(), items().
  - Перші двоє повертають ітератор по всіх ключах та значеннях у словнику відповідно, який можна використовувати як список.
  - Метод items() повертає ітератор по парах «ключ-значення» у вигляді кортежів (key, value) для кожної пари у словнику.

```
>>> myDictionary = {"key1": "value1", "key2":"value2", "key3": "value3"}
```

```
>>> keys = myDictionary.keys()
>>> print(type(keys))
<class 'dict_view'>
>>>
>>> print(list(keys))
['key2', 'key3', 'key1']
```

```
>>> values = myDictionary.values()
>>> print(type(values))
<class 'dict_view'>
>>>
>>> print(list(values))
['value2', 'value3', 'value1']
```

```
>>> items = myDictionary.items()
>>> print(type(items))
<class 'dict_view'>
>>> print(list(items))
[('key2', 'value2'), ('key3', 'value3'), ('key1', 'value1')]
>>>
>>>
>>> len(myDictionary)
3
```

Приклад з виведенням вмісту словника stock

```
>>> for stock, values in stocks.items():
...     print("{} last value is {}".format(stock, values[0]))
...
GOOG last value is 613.3
BBRY last value is 10.50
MSFT last value is 30.25
```

- Для отримання даних можна використовувати квадратні дужки, методи get(), setdefault() або ітерувати по словнику методом items().
- Задавання значень потребує того ж синтаксису квадратних дужок:

```
>>> stocks["GOOG"] = (597.63, 610.00, 596.28)
>>> stocks['GOOG']
(597.63, 610.0, 596.28)
```

Можна використовувати різнотипні ключі в межах одного словника

```
random_keys = {}
random_keys["astring"] = "somestring"
random_keys[5] = "aninteger"
random_keys[25.2] = "floats work too"
random_keys[("abc", 123)] = "so do tuples"

class AnObject:
    def __init__(self, avalue):
        self.avalue = avalue

my_object = AnObject(14)
random_keys[my_object] = "We can even store objects"
my_object.avalue = 12
try:
    random_keys[[1,2,3]] = "we can't store lists though"
except:
    print("unable to store list\n")

for key, value in random_keys.items():
    print("{} has value {}".format(key, value))
```

- Зазвичай ключі є рядками.
  - Але можуть бути і кортежами, числами та ін. об'єктами.
- Оскільки списки допускають зміну значень у будь-який момент, їх не можна хешувати в конкретне значення.

Хешовані об’єкти

- Визначають алгоритм перетворення об’єкту в унікальне ціле число, доречне для швидкого перегляду.
  - При пошуку переглядається саме хеш-значення.
  - Об’єкти, що вважаються рівними, повинні мати однаковий незмінний хеш.
  - Списки та словники – змінювані, тому не можуть бути ключами.
- Значення у парі «ключ-значення» може бути довільним.
  - Наприклад, можна використовувати рядковий ключ, що відображається на список, або вкладати словник у словник.

Використання словників

- 1) всі ключі представляють різні об’єкти.
  - Так працює система індексації.
  - Значення у парі з ключем можуть бути складними об’єктами.
- 2) кожен ключ представляє певний аспект єдиної структури;
  - Ймовірно, для кожного об’єкта буде власний словник, набори ключів будуть схожими, проте не ідентичними.
  - Може представлятись також за допомогою іменованих кортежів.
  - Іменовані кортежі краще використовувати, якщо точно відомо потрібні атрибути для зберігання даних, причому всі їх частини потрібно постачати одразу при конструюванні елементу.
  - Словник доречніший тоді, коли потрібно створювати чи змінювати ключі протягом часу або невідома точна кількість можливих ключів.

Використання defaultdict

- При кожному доступі до словника слід перевіряти, чи є вже таке значення в ньому; якщо ні – встановити 0 за замовчуванням.
  - Для цього можна використовувати іншу версію словника - defaultdict:

```
def letter_frequency(sentence):
    frequencies = {}
    for letter in sentence:
        frequency = frequencies.setdefault(letter, 0)
        frequencies[letter] = frequency + 1
    return frequencies
```

```
from collections import defaultdict
def letter_frequency(sentence):
    frequencies = defaultdict(int)
    for letter in sentence:
        frequencies[letter] += 1
    return frequencies
```

- Якщо буква не представлена в defaultdict, повертається число 0 при спробі доступу.
  - Потім додаємо 1 до числа, щоб показати, що знайдено екземпляр літери.
  - Наступного разу це число буде ненульове.

Огляд коду

- defaultdict корисний при створенні словників контейнерів.
  - Для створення словника біржових цін за останні 30 днів можна використовувати символ біржі та список цін;
  - При першому доступу до біржі список цін буде порожнім.
  - З одним ключем можна асоціювати множини або навіть порожні словники.
- Можна створювати власні функції та передавати їх у defaultdict
  - Нехай у defaultdict кожен новий елемент міститиме кортеж з кількостей доданих у цей момент до словника елементів, а також порожній список для інших речей.

```
from collections import defaultdict
num_items = 0
def tuple_counter():
    global num_items
    num_items += 1
    return (num_items, [])

d = defaultdict(tuple_counter)
```

Можна отримати доступ до порожніх ключів та вставити у список за один підхід:

```
>>> d = defaultdict(tuple_counter)
>>> d['a'][1].append("hello")
>>> d['b'][1].append('world')
>>> d
defaultdict(<function tuple_counter at 0x82f2c6c>,
{'a': (1, ['hello']), 'b': (2, ['world'])})
```

Counter

```
from collections import Counter

responses = [
    "vanilla",
    "chocolate",
    "vanilla",
    "vanilla",
    "caramel",
    "strawberry",
    "vanilla"
]
print(
    "The children voted for {} ice cream".format(
        Counter(responses).most_common(1)[0][0]
    )
)
```

- Попередній код, який підраховував кількість символів у рядку, можна записати в одному рядку:

```
from collections import Counter
def letter_frequency(sentence):
    return Counter(sentence)
```

- Екземпляр класу Counter працює як прокачаний словник, у якому ключі є символами для підрахування, а значення – їх відповідна кількість.
  - Один з найбільш поширених методів – most_common(): повертає список кортежів (key, count), упорядкований за кількістю.
  - Опційно можна передати ціле число в метод most_common(), щоб отримати тільки найбільш розповсюджені елементи.

Списки

- Не потребують імпортування, їх методи рідко викликаються.
  - Доступне ітерування по списку без явного звернення до об’єкта-ітератора, для конструювання списку наявний спеціальний синтаксис.
- Зазвичай використовуються за потреби збереження кількох екземплярів «того ж» типу;
  - Списки мають впорядковану структуру, зазвичай за черговістю вставки або за результатами сортування.
  - Списки доречні за потреби внесення змін у їх вміст.
- Не використовуйте списки для збирання різних атрибутів від окремих елементів.
  - Доречніші кортежі, іменовані кортежі, словники та об’єкти.
  - У деяких мовах може створюватись списки з елементами різних типів; наприклад, ['a', 1, 'b', 3].
  - Це значно ускладнить доступ до таких елементів, тому краще цю можливість не використовувати.
  - Пов’язані елементи можна групувати за допомогою словника або списку кортежів.

Приклад з обчисленням частот на базі списків

- Код набагато складніший, ніж для словників:

```
import string
CHARACTERS  = list(string.ascii_letters) + [" "]

def letter_frequency(sentence):
    frequencies = [(c, 0) for c in CHARACTERS]
    for letter in sentence:
        index = CHARACTERS.index(letter)
        frequencies[index] = (letter,frequencies[index][1]+1)
    return frequencies
```

  - Доводиться постійно оновлювати index для списку frequencies, створюючи новий кортеж та відкидаючи стару версію.
  - Такий код складно читати, він вимагає багато пам’яті та постійної роботи збирача сміття!

Поширені методи (функції) для роботи зі списками

- append(element) – додає елемент у кінець списку
- insert(index, element) – вставляє елемент в задану позицію в списку
- count(element) – обчислює, скільки разів елемент зустрічається в списку
- index() – вказує індекс елемента зі списку, викидає виняток, якщо не знаходить цей елемент
- find() – аналог index(), проте повертає -1, якщо не знаходить елемент
- reverse() – змінює порядок елементів від останнього до першого
- sort() – займається сортуванням елементів списку

Множини

- Списки недоречні при потребі в підтримці унікальності елементів у наборі даних.
  - Наприклад, плейліст може містити багато пісень одного виконавця.
  - При необхідності відсортувати треки за виконавцем потрібно буде перевіряти, чи додавали виконавця до переліку раніше.
- У мові Python множини (sets) можуть містити будь-який хешований об’єкт.
  - Це ті ж об’єкти, що й ключі в словниках.
  - Аналогічно до математичних множин, вони можуть містити лише одну копію кожного об’єкта.
  - Тому для сортування краще замінити список виконавців на множину.

```
song_library = [("Phantom Of The Opera", "Sarah Brightman"),
        ("Knocking On Heaven's Door", "Guns N' Roses"),
        ("Captain Nemo", "Sarah Brightman"),
        ("Patterns In The Ivy", "Opeth"),
        ("November Rain", "Guns N' Roses"),
        ("Beautiful", "Sarah Brightman"),
        ("Mal's Song", "Vixy and Tony")]
```

```
artists = set()
for song, artist in song_library:
    artists.add(artist)

print(artists)
```

Як і словники, множини невпорядковані

- Робота програми:

```
>>> "Opeth" in artists
True
>>> for artist in artists:
...     print("{} plays good music".format(artist))
...
Sarah Brightman plays good music
Guns N' Roses plays good music
Vixy and Tony play good music
Opeth plays good music
>>> alphabetical = list(artists)
>>> alphabetical.sort()
>>> alphabetical
["Guns N' Roses", 'Opeth', 'Sarah Brightman', 'Vixy and Tony']
```

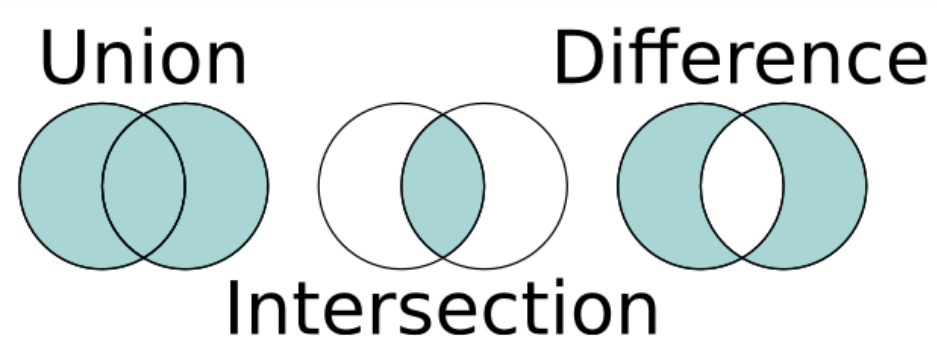
Унікальність елементів множин не є основною ціллю їх створення

- Множини найбільш корисні, якщо комбінуються між собою.

```
my_artists = {"Sarah Brightman", "Guns N' Roses",
        "Opeth", "Vixy and Tony"}

auburns_artists = {"Nickelback", "Guns N' Roses",
        "Savage Garden"}

print("All: {}".format(my_artists.union(auburns_artists)))
print("Both: {}".format(auburns_artists.intersection(my_artists)))
print("Either but not both: {}".format(
    my_artists.symmetric_difference(auburns_artists)))
```




```
All: {'Sarah Brightman', "Guns N' Roses", 'Vixy and Tony',
'Savage Garden', 'Opeth', 'Nickelback'}
Both: {"Guns N' Roses"}
Either but not both: {'Savage Garden', 'Opeth', 'Nickelback',
'Sarah Brightman', 'Vixy and Tony'}
```

Метод difference()

- Різниця множин повертає всі їх елементи, які є в одній множині (викликаючій), проте відсутні в іншій.
  - Операція працює як половина симетричної різниці та може представлятись в коді за допомогою оператора «-».

```
my_artists = {"Sarah Brightman", "Guns N' Roses",
        "Opeth", "Vixy and Tony"}

bands = {"Guns N' Roses", "Opeth"}

print("my_artists is to bands:")
print("issuperset: {}".format(my_artists.issuperset(bands)))
print("issubset: {}".format(my_artists.issubset(bands)))
print("difference: {}".format(my_artists.difference(bands)))
print("*"*20)
print("bands is to my_artists:")
print("issuperset: {}".format(bands.issuperset(my_artists)))
print("issubset: {}".format(bands.issubset(my_artists)))
print("difference: {}".format(bands.difference(my_artists)))
```

```
my_artists is to bands:
issuperset: True
issubset: False
difference: {'Sarah Brightman', 'Vixy and Tony'}
********************
bands is to my_artists:
issuperset: False
issubset: True
difference: set()
```

ДЯКУЮ ЗА УВАГУ!

Наступне питання: організація Python-коду за допомогою функцій