Міністерство освіти і науки України
Львівський національний університет імені Івана Франка

**Стахіра Р. Й., Кулик П. Р.,
Шувар Р. Я.**

Kotlin для роботи з даними

Навчальний посібник

Львів
2023

УДК 004.43.04(075.8)
С 78

Рецензенти:
д-р фіз. мат. наук, проф. *П. С Венгерський*
(Львівський національний університет імені Івана Франка);
канд. техн. наук, доц. *О .Я. Олар*
(Чернівецький національний університет
імені Юрія Федьковича);
канд. техн. наук, доц. *О. Ю. Рощупкін*
(Компанія GlobalLogic);

Рекомендовано Вченою радою факультету
електроніки та комп'ютерних технологій
Львівського національного університету імені Івана Франка
Протокол № 36/23 від 29 травня 2023 р.

Стахіра Р. Й.
С 78 Kotlin для роботи з даними : навч. посібник / Р. Й. Стахіра, П. Р. Кулик, Р. Я. Шувар. – Львів. 2023. – 160 с.

Розглянуто основні концепції мови програмування Kotlin та її використання у галузі «Наука про дані». Коротко описано основні конструкції мови і головні концепції створення додатків мовою Kotlin. Розглянуто основи роботи з даними в інтерактивних редакторах та методи використання існуючих бібліотек для мови Kotlin. Усі головні положення розглянуто з використанням навчальних прикладів та ілюстровано графічними матеріалами.

Посібник буде корисним для студентів усіх спеціальностей напряму підготовки «Інформаційні технології».

©Стахіра Р. Й., Кулик П. Р., Шувар Р. Я.
©Львівський національний університет
імені Івана Франка, 2023

Зміст

Вступ **5**

1. Основи Kotlin **7**

1.1. Ознайомлення з Kotlin 7
1.2. Цикли та умови 18
1.3. Функції в Kotlin 21
1.3.3. Створення функцій. Аргументи: визначені (іменовані) та за замовчуванням 21
1.3.3. Функції вищого порядку. Лямбда-функції. Вбудовані функції. 23
1.3.3. Деструктуризація, varargs та spread-оператор 28
1.4. Класи та об'єкти 30
1.4.4. Властивості класу. Наслідування. Модифікатори доступу 33
1.4.4. Інтерфейси та абстрактні класи. Data-класи. Sealed-класи. Enum-класи 38
1.4.4. Вкладені та внутрішні класи 42
1.4.4. Узагальнення 43
1.4.4. Делегати та делеговані властивості. Лінива ініціалізація. 44
1.5. Інші концепти 49
1.5.5. Null-безпека та приведення типів 49
1.5.5. Порівняння в Kotlin 51
1.5.5. Ключове слово this 52
1.5.5. Перевизначення операторів 53
1.5.5. Функції-розширення 54
1.5.5. Функції області видимості 54
1.6. Колекції в Kotlin 57

1.6.6. Загальний огляд. Ітератори. Діапазони. Послідовності 57
1.6.6. Масиви, списки 64
1.6.6. Основні операції: перетворення, фільтрація, вибірка, групування 66
1.6.6. Пошук, сортування та об'єднання колекцій . 71
1.7. Багатопотоковість 75
1.7.7. Створення потоків у Kotlin 75
1.7.7. Корутини (Coroutines) 77

2. Kotlin-ресурси для роботи з даними **82**
2.1. Інтерактивні редактори для роботи з даними 82
2.2. Бібліотеки 91
2.2.2. Lets-Plot: графіка для статистичних даних . 91
2.2.2. Kravis: бібліотека для візуалізації табличних даних 104
2.2.2. Multik: бібліотека багатовимірних масивів для Kotlin 109
2.2.2. Kotlin-Statistics: математичні та статистичні розширення для Kotlin 116
2.2.2. Krangl: маніпулювання даними 125
2.2.2. KotlinDL: API глибинного навчання високого рівня в Kotlin 136

Список використаних джерел **144**

A. Приклади програм **147**
A.1. Використання бібліотек Kotlin-Statistics та Lab-Plot 147
A.2. Використання бібліотек Krangl та Kravis 151

Предметний покажчик **155**

Вступ

Ідея створення Kotlin зародилася у JetBrains у 2010 році. На той час компанія вже була визнаним виробником інструментів розробки для багатьох мов, включаючи Java, C#, JavaScript, Python, Ruby та PHP. IntelliJ IDEA флагманський продукт, інтегроване середовище розробки (IDE) для Java включає плагіни для Groovy, Scala, Python та багатьох інших мов.

У 2011 році компанія JetBrains анонсувала розробку мови програмування Kotlin як альтернативу мовам Java та Scala, код якої також виконується під управлінням віртуальної машини Java (Java Virtual Machine). Через шість років компанія Google анонсувала початок офіційної підтримки Kotlin, як мови для розробки під операційну систему Android. З травня 2019 року є рекомендованою мовою програмування для розробки Android застосунків.

Область застосування Kotlin швидко зросла від мови зі світлим майбутнім до мови підтримки програм провідної світової операційної системи. Сьогодні великі компанії на кшталт Google, Uber, Netflix, Capital One, Amazon та інші офіційно використали Kotlin за його зручність, зрозумілий синтаксис, сучасні функції та повну сумісність з Java.

Отже, Kotlin – це прагматична, статично типізована мова, яка підтримує написання коду як в об'єктно орієнтованому, так і у функціональному стилі, а також компілюється в різні платформи: JVM, JS, Native. Завдяки статичній типізації, Kotlin більш продуктивний у великих проєктах, ніж Python, а також дає змогу уникнути помилок виконання ще на етапі програмування.

Незважаючи на відносно юний вік, Kotlin вже має велику еко-

систему бібліотек, фреймворків та інструментів, більшість з яких створено зовнішньою спільнотою розробників, не залишили без уваги такий напрям, як «Наука про дані» (Data Science).

Структура видання

Посібник «Kotlin для роботи з даними» розповість про мову Kotlin і як писати на ній програми. Перший розділ присвячений огляду основних особливостей мови Kotlin, поступово розкриваючи найбільш відмінні аспекти, такі як підтримка створення високорівневих абстракцій та предметно-орієнтованих мов (Domain-Specific Languages, DSL).

Головною метою другого розділу є огляд інструментів для роботи з даними, до яких належать інтерактивні редактори Jupyter Notebook і Apache Zeppelin та багато бібліотек. Функціонал бібліотек, що описані в цьому огляді, охоплює такі напрями, як машинне навчання, візуалізація та маніпуляції даних, статистика та ін.

Сподіваємось, що посібник допоможе Вам упровадити Kotlin у поточне робоче середовище та стане корисним інструментом для роботи в галузі «Наука про дані».

Розділ 1.

Основи Kotlin

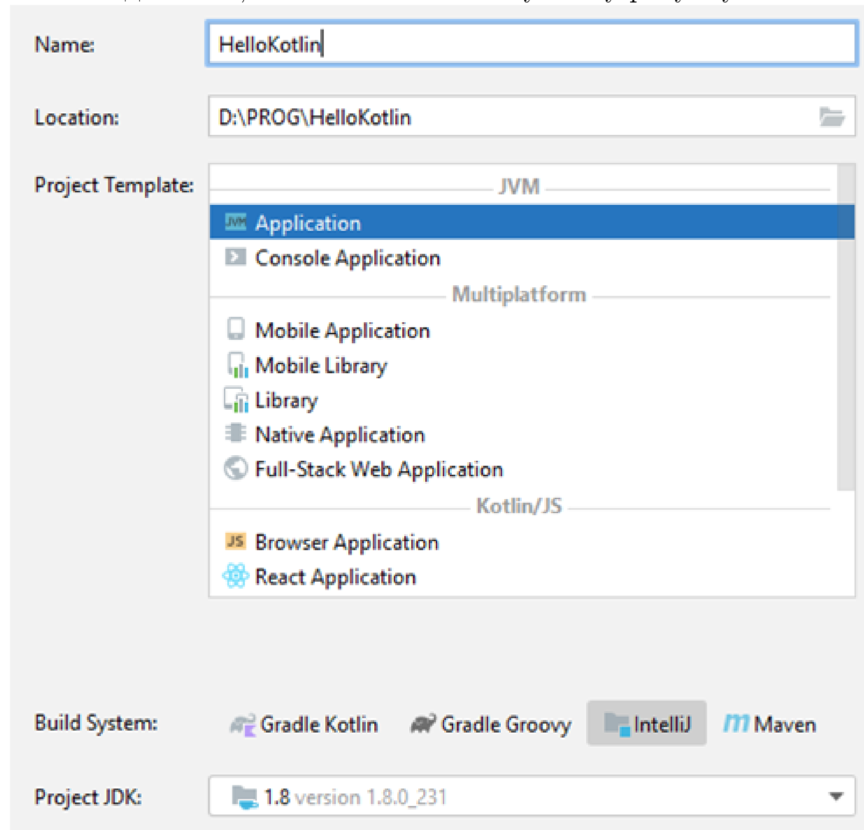
1.1. Ознайомлення з Kotlin

Kotlin – популярна мова програмування, створена командою JetBrains та рекомендована Google як офіційна мова для Android розробки. Код, написаний на Kotlin, компілюється у байт код Java Virtual Machine (Kotlin/JVM) та має підтримку основних платформ, таких як: Linux, macOS, Windows, iOS тощо (Kotlin/Native). Крім того, код Kotlin можна транслювати у JavaScript (Kotlin/JS).

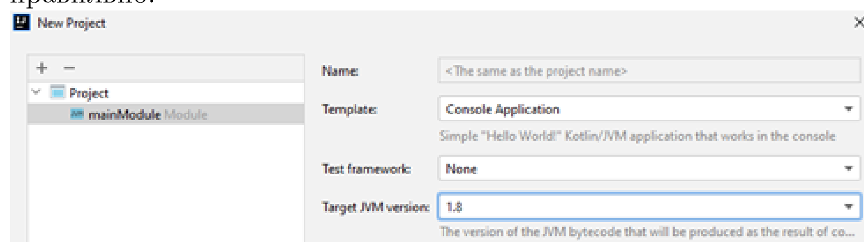
Хоча Kotlin синтаксично відрізняється від Java, семантично він досить подібний, тому, якщо Ви вже маєте досвід розробки на Java, впоратися з Kotlin не стане проблемою. З Kotlin доведеться писати менше та компактніше, оскільки він пропонує безліч засобів та функцій для ефективного програмування, не витрачаючи час на написання лишнього коду. З усіма наведеними особливостями та засобами Ви ознайомитесь у цьому посібнику.

Налаштування проєкту в Intellij Idea та перша програма. Для створення нового Kotlin проєкту, Вам потрібно обрати в Intellij Idea тип проєкту Kotlin (у деяких випадках потрібно попередньо встановити відповідний плагін для Kotlin підтримки), вказати назву, шлях, шаблон як консольний застосунок, обрати тип побудови як Intellij (простий варіант), та вибрати конкретну версію JDK із запропонованих. Загалом вікно конфігурації повин-

но виглядати так, як показано на наступному рисунку.



Після переходу у наступне вікно створення обираємо шаблон як консольний застосунок та впевнюємось, що всі опції виставлені правильно.



Щойно проєкт створено, можемо простежувати у ньому файл main.kt (src/main/kotlin/main.kt) із наступною функцією.

```
fun main(args: Array<String>) {
    println("Hello World!")
}
```

Запустивши цей код, отримуємо повідомлення в консолі змісту:

```
Hello World!
```

Як бачимо з прикладу, функція в Kotlin оголошується, починаючи із ключового слова `fun`. В дужках указуємо список параметрів, які функція прийматиме. У нашому випадку це змінна із назвою `args`, після чого йде розділювач та вказаний тип цієї змінної – `Array<String>`. Для виводу інформації в консоль використовуємо функцію `println()`.

Готово! Ви успішно створили Kotlin проєкт та запустили першу консольну програму. В наступному розділі детальніше розглянемо основи синтаксису та нові можливості, що надає ця мова програмування.

Особливості синтаксису та базові принципи. *Змінні* у Kotlin визначають за допомогою ключових слів `val` та `var`. За допомогою `val` можна створити змінну лише для читання (початкове значення задають під час ініціалізації), водночас як `var` можна змінювати згодом.

```
val count1 = 10                         //змінна тільки
↪  для читання, її тип  визначається автоматично (Int)
val count2: Int = 20        //змінна для читання, її тип
↪  вказано явно
count1 = 35                                 //помилка,
↪  змінити значення не можливо,

var index = 0                 //змінна для читання та
↪  запису
index = 15                    //присвоєння нового
↪  значення
```

Функції в Kotlin створюють за допомогою ключового слова `fun`, після чого йде назва, список параметрів та тип, який вона повертає.

```
        fun multiplication(a: Int, b: Int): Int {
                return a * b
}
```

Варто зазначити, що змінні `a` та `b` є лише для читання, тобто змінити їх значення в тілі функції неможливо. Також як тіло функції можна використати вираз. Тип значення, що повертатиметься, можна вказати явно, або залишити для автоматичного визначення:

```
fun multiplication(a: Int, b: Int): Int = a * b
↪            //значення вказано явно

fun multiplication(a: Int, b: Int) = a * b
↪            //значення визначається автоматично
```

Якщо ж функція нічого не повертає, як тип можна вказати `Unit`, або пропустити його:

```
fun printHello(){
            println("Hello!")
}

fun printHello(): Unit{
            println("Hello!")
}
```

Умовний оператор `if-else`, конструкція `when`.

Для перевірки умови використовують оператор `if(умова)`

```
val str1: String  = "one"
if(str1 == "two")                //перевірка умови
        println("Yes")           //якщо умову виконано
```

```
                    else
                println("No")     //якщо умову не
↪   виконано
```

Оператор `if` також може бути як вираз та повертати значення.

```
val str1: String = "one"
val result: String = if(str1 == "two") "Yes" else "No"
```

Вираз `when` у Kotlin чимось нагадує `switch` в Java. Наприклад, виведемо текст залежно від того, яке значення матиме змінна `cipher`.

```
val cipher = 1
when(cipher){
        1 -> println("Один")
        2 -> println("Два")
        3 -> println("Три")
        else -> println("Інше")
}
```

Ми також можемо використовувати його для отримання значення з умови та зручно перевіряти відношення аргументу до певного типу.

```
fun getAnyType(input: Any) = when(input) {
                is Number -> "Це число"
                is String -> "Це рядок"
                else -> "Це щось інше"
        }
```

У цьому прикладі функція поверне потрібне значення залежно від того, якого типу буде вхідний аргумент `input`.

try-catch конструкція також може повертати значення виразу.

```
val result = try{
        //повертаємо значення
        true
} catch (e: Exception) {
        //TODO
        false
}
```

Коментарі та шаблони рядків. Існує два типи коментарів: однорядковий та блочний.

```
//коментар-рядок
/*
коментар-блок
*/
```

Шаблони рядків – одна зі зручних можливостей Kotlin. Вона дає можливість без особливих зусиль вставляти в рядок значення змінних або виразів.

```
val a = 1
val b = 2
println("Це приклад стандартного виводу. Значення
 ↪  змінної a=" + a + " та b=" + b)
println("Це приклад шаблону рядка. Значення змінної a=$a
 ↪  та b=$b")
println("Це приклад шаблону рядка з виразом. Значення
 ↪  суми ${a + b}")
```

Цикли та інтервали. Цикл `for` можна використовувати для ітерації по елементах списку та по індексах.

```
val list = listOf("foo", "bar")

//ітерація по елементах
for(el in list) {
```

```
        println(el)
}

//ітерація по індексах
for(index in list.indices) {
        println(list[index])
}
```

Цикл `for` також можна застосовувати з інтервалами.

```
//ітерація від 0 до 10 включно
for(i in 0..10)
        println(i)
//ітерація від 0 до 10 не включно
for(i in 0 until 10)
        println(i)
//ітерація від 0 до 10 не включно із кроком 2
for(i in 0 until 10 step 2)
        println(i)
//ітерація від 10 до 0
for(i in 10 downTo 0)
        println(i)
```

Цикл `while` використовують в класичному варіанті.

```
//ітерація від 10 до 1 за допомогою while
var index = 10
while(index > 0) {
        println(index)
        index--
}
```

Колекції у Kotlin можна визначати як лише для читання, так і з можливістю розширення. Нижче наведено варіанти створення колекцій:

```
//список рядків лише для читання
val immutableList = listOf("foo", "bar")
//список рядків котрий можна змінювати
val mutableList = mutableListOf("foo", "bar")
mutableList.add("next")
```

У колекцію immutableList (тип `List`) ми не можемо додати нові елементи, водночас як mutableList (тип `MutableList`) без проблем цю можливість надає.

Null-безпека. Під час вивчення Kotlin, Ви можете задати питання, що означає знак ? біля типу змінної. Знак ?, що стоїть біля типу змінної, означає, що ця змінна може бути `null` (відсутність значення).

```
var str1: String = "Foo"
str1 = null //помилка, ця змінна не може містити null

var str2: String? = "Bar"
str2 = null //помилка відсутня, оскільки біля типу
↪  змінної стоїть ?
```

Це не єдине місце, де ми можемо використати ?. За його допомогою можна створити ланцюг викликів: якщо на одному з елементів зустрінеться `null` – виконання буде зупинено. Подивимось на код нижче.

```
val str: String? = null
println(str?.length)
```

Виконавши цей код, отримуємо вивід `null` замість `NullPointerException`.

Типи даних. Цілочисельні типи Kotlin

`Byte` – 8 біт, від -128 до 127

`Short` – 16 біт, від -32 768 до 32 767

`Int` – 32 біта, від -2 147 483 648 до 2 147 483 647

`Long` – 64 біта, від -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807

Приклад використання
```
val bValue: Byte = 1
val sValue: Short = 1
val iValue: Int = 1_450_000_000
val lValue: Long = 450_000_000_000
val autoInt = 10000000                    //автоматично
↪  визначено тип як Int
val autoLong = 1000000000000          //автоматично
↪  визначено тип як Long
```

Беззнакові цілочисельні типи (те саме, що Unsigned)

`UByte` – 8 біт, від 0 до 255

`UShort` – 16 біт, від 0 до 65 535

`UInt` – 32 біта, від 0 до 4 294 967 295

`ULong` – 64 біта, від 0 до 18 446 744 073 709 551 615

Приклад використання
```
val uBValue: UByte = 1u
val uSValue: UShort = 1u
val uIValue: UInt = 1_450_000_000u
val uLValue: ULong = 450_000_000_000u
```

Числа з плаваючою крапкою

`Float` – 32 біта

`Double` – 64 біта

Згідно зі стандартом IEEE 754, числа типу `Float` забезпечують одинарну точність, `Double` - подвійну.

Приклад використання
```
val fValue: Float = 5.3453253f
val dValue: Double = 10.15645479541253
val autoFloat = 1.7f           //автоматично визначено
↪  тип як Float
```

```
val autoDouble = 154.8          //автоматично
↪  визначено тип як Double
```

Логічний тип – `Boolean` приймає два значення: `true` (умова правильна) та `false` (умова неправильна). Крім цього, для нього доступні операції порівняння, такі як: `&&` – ліниве логічне "і"; `||` – ліниве логічне "або"; `!` – заперечення.

Приклад використання

```
val bTrueValue = true
val bFalseValue = false

//умова не виконується оскільки bFalseValue = false
if(bTrueValue && bFalseValue)
        println("Hello 1!")

//умова виконується
if(bTrueValue && !bFalseValue)
        println("Hello 2!")
```

Символьний тип подано типом `Char`. Значення такого типу беруть в одинарні лапки `'_'`.

Приклад використання

```
val cValue = 'b'
val nValue = '\n'

println(nValue)
println(cValue)
```

Рядки подано типом `String`. Значення такого типу беруть у подвійні лапки `""`. Такий формат підтримує екранування, наприклад, перехід на новий рядок `'\n'` тощо. Якщо помістити текст у потрійні лапки `"""`, наш рядок може містити будь-які символи, проте не буде підтримувати екранування.

Приклад використання

```
val simpleString = "Hello world!"
val simpleStringLiteral1 = "Hello world!\nHow are you?"
val simpleStringLiteral2 = """We can use here "this
↪  sort of text style" as well \n"""

println(simpleString)
println(simpleStringLiteral1)
println(simpleStringLiteral2)
```

Результат виконання (як бачимо, `'\n'` у цьому випадку не інтерпретоване):

```
Hello world!
Hello world!
How are you?
We can use here "this sort of text style" as well \n
```

Масиви. Для подання масивів у Kotlin існує клас `Array`. Доступ до елементів масиву можна виконувати звиклим способом `[]`. Для оголошення масиву використовують виклик `arrayOf`. Крім того, у Kotlin є готові класи для представлення масивів типу `ByteArray`, `ShortArray`, `IntArray`, `LongArray`, `BooleanArray`, `DoubleArray`, `FloatArray`, `CharArray`.

Приклад використання

```
val arrayInt = arrayOf(1, 7, 9, 2)
val arrayBool1 = arrayOf(true, false)

//за допомогою конструктора класу
//заповнюємо масив розміром 5 елементами, значення яких
↪  відповідає їх індексу
val newArrayInt = Array(5) { index -> index }

val arrayByte = byteArrayOf(1, 4, 125)
val arrayLong = longArrayOf(1900000000000, 456000000)
```

```
val arrayBool2 = booleanArrayOf(false, true)

val arrayFloat = floatArrayOf(1f, 3.1f)
val arrayDouble = doubleArrayOf(1.3, 5.4)
val charArray = charArrayOf('a', 'b', 'c')

//доступ/заміна елементів
val first = arrayInt[0]
arrayInt[0] = 5
```

1.2. Цикли та умови

Умови. Як було описано в попередніх розділах, умовний оператор `if` можна використовувати також як вираз та повертати значення цього виразу. Проте він також може містити блок коду, що в кінці повертатиме потрібне значення:

```
val something = "foo"
val result = if(something == "foo"){
                println("Something is foo")
                "good"
        }else {
                println("Something is not foo")
                "bad"
        }
//в result передається останній рядок блоку коду як
↪  значення виразу
```

Конструкцію `when` також можна використовувати як для більш зручного аналога конструкції `switch`, котрий можна бачити у Java. Як і `if`, `when` аналогічно може повертати значення виразу. Під час використання `when` перевіряється кожна з умов, вказаних у тілі конструкції, якщо одна з них виконується – всі інші відкидаються.

```
val str = "one"
val result = when(str){
        "one" -> 1
        "two" -> 2
        "three" -> 3
        else -> -1
        }
```

У наведеному прикладі ми порівнюємо значення рядка `str` із кожною умовою, та повертаємо значення у змінну `result`. Зверніть увагу, що у цьому випадку потрібно обов'язково вказати `else`. Пропустити `else` можна лише у випадку, коли ми вказали усі можливі умови (наприклад, під час використання `enum`). Якщо є декілька можливих варіантів для однієї умови, є можливість перечислити їх через кому:

```
val foodType = FoodType.HEALTHY
when(foodType) {
        FoodType.HEALTHY -> println("Healthy")
        FoodType.FAST, FoodType.FAT -> println("Bad")
        }
```

У цьому випадку було використано усі варіанти `enum`-класу FoodType, тому else використовувати, не потрібно. when також можна використовувати не передаючи в нього аргумент. Тоді це служитиме як простий `if-else` ланцюг (в цьому прикладі використовувати `else` необов'язково).

```
val data = "Some data"
//аргумент відсутній
when{
        data.length > 5 -> println("Data length is > 5")
        data.isEmpty() -> println("Data is empty")
        else -> println("Data: $data")
        }
```

Цикли. У попередньому підрозділі було наведено приклади застосування циклу `for`. Застосувати його можна до будь-якого об'єкта, що має реалізовану внутрішню або зовнішню функцію `iterator`, яка повертає `Iterator<>`.

```
val items = listOf("a", "b", "c", "d", "e")
for(item in items){
        println(item)
        }
```

Також зручно використовувати прохід з індексом

```
val items = listOf("a", "b", "c", "d", "e")
for((i, item) in items.withIndex()){
        println("Index: $i Item: $item")
        }
```

У Kotlin також існує цикл `do-while`, про який раніше не згадували. Відрізняється він від звичайного `while` тим, що спочатку виконує своє тіло, а лише потім перевіряє умову. В циклах також можна використовувати оператори переходу break (завершує найближчий цикл) та `continue` (перериває поточну ітерацію найближчого циклу та переходить на наступну):

```
val items = listOf("a", "b", "c", "d", "e", "f", "g")
outer@ for(i in 0..5) {
        for (item in items) {
        //пропускаемо поточну ітерацію та переходимо до
↪  наступної
                if (item == "c")
                        continue
                //завершуємо  зовнішній цикл за
↪  допомогою мітки outer@
                if (item == "f")
                        break@outer
                println("Item: $item")
```

```
                }
        }
```

У наведеному прикладі для переходу у зовнішній цикл використовують мітку `outer@`, під час виклику `break@outer` ми завершуємо зовнішній цикл, а не найближчий внутрішній.

1.3. Функції в Kotlin

Створення функцій у Kotlin значно відрізняється від створення методів у Java. Kotlin дає можливість робити це просто та гнучко, минаючи багато непотрібних дій з боку програміста. В цьому розділі розглянемо створення функцій, їхні різновидності та застосування.

1.3.3. Створення функцій. Аргументи: визначені (іменовані) та за замовчуванням

Створити функцію у Kotlin можна за допомогою ключового слова `fun`, після чого йде назва, список параметрів та тип, що функція повертає.

```
fun div(a: Double, b: Double): Double {
        return a / b
}
```

Виклик функції відбувається аналогічно, як у Java:

```
val result = div(4.0, 2.0)
```

Параметри функції можна задати за замовчуванням, тоді не обов'язково передавати аргумент під час виклику функції.

```
fun div(a: Double, b: Double = 2.0): Double {
        return a / b
}
```

```
//за замовчуванням значення другого аргументу буде  2.0
val result = div(4.0)
```

Крім того, якщо функція має багато параметрів, можна використати іменовані аргументи, тобто передавати аргументи під час виклику функції в довільному порядку, вказуючи їх ім'я.

```
fun doSomething(first: Double, second: Double, third:
↪  Double, fourth: Double, fifth:Double) {
//todo
}
//Виклик
doSomething(second = 2.0, fifth = 4.0, first = 20.0,
↪  third = 5.0, fourth = 5.0)
```

Як бачимо з прикладу, якщо функція нічого не повертає, вказувати тип `Unit` не обов'язково. Якщо функція містить лише один вираз, або вираз-умову, це також можна скоротити:

```
fun div(a: Double, b: Double = 2.0) = a / b

fun diff(a: Double, b: Double, abs: Boolean) = if(abs)
↪  abs(a - b) else a - b
```

Існує також такий досить зручний тип функцій, як `infix` функції. Вони позначаються спочатку ключовим словом `infix`. Для таких функцій можна опустити запис крапки при виклику та дужок при передачі аргументів:

```
infix fun Double.diff(value: Double) = this - value
//Виклик
println(10.5 diff 5.0) //отримаємо 5.5
```

У цьому прикладі наведено використання інфіксної функції. Як бачимо, створити таку функцію можна лише у випадку розширення (або члена іншої функції): з одним параметром та можливістю приймати лише один аргумент. Оголошувати функції

можна за межами класу (Kotlin файл), як члени класу, та всередині інших функцій (локальні):

```
//глобальна функція
fun getValue() = 10

fun main(args: Array<String>) {
        //локальна функція
    fun sum(a: Int, b: Int) = a + b
        //вивід значень
            println(getValue())
            println(sum(10, 10))
            println(Vehicle().speed())
}

class Vehicle{
        //функція-член класу
    fun speed() = 10.5
}
```

Крім того, у функціях можна використовувати параметризовані типи (Generics).

```
fun <T: Number> asValueString(input: T) = "Value:
↪  $input"
```

1.3.3. Функції вищого порядку. Лямбда-функції. Вбудовані функції.

Загалом функцією вищого порядку у Kotlin можна назвати таку, що може приймати іншу функцію як параметр, або ж повертати її як результат виразу. Розглянемо приклад:

```
fun <T> mutate(list: MutableList<T>, mutator: (T) -> T){
        val it = list.listIterator()
        while(it.hasNext()){
```

```
            val oldValue = it.next()
        val newValue = mutator(oldValue)
        if(oldValue !== newValue)
                it.set(newValue)
        }
}
```

Отже, як бачимо з прикладу, тут маємо два параметри: параметризований список `list` (тобто список, на цей момент, це список елементів довільного типу) та функціональний тип `mutator`. Функціональний тип це функція, яка на вхід приймає елемент списку, та повертає модифікований елемент цього ж типу. Функція, що продаватиметься за викликом функції вищого порядку `mutate` в якості аргументу називається лямбда-функцією. З цього прикладу бачимо, що у функціонального типу є список параметрів (у цьому випадку це лише параметр `T`) та тип, що повертається та йде після стрілки ->. Сам лямбда-вираз завжди оточений `{}`.

```
val items = mutableListOf("a", "b", "c", "d", "e")
mutate(items, { item -> "-$item" } )
println(items)
// Вивід:        [-a, -b, -c, -d, -e]
```

Як бачимо з результату, під час виклику функції вищого порядку `mutate`, ми для кожного елементу списку (`item`) викликали лямбда-функцію, додавши префікс "-" до кожного значення. Якщо функціональний тип є останнім із параметрів, під час виклику функції його можна винести за дужки.

```
mutate(items) { item -> "-$item" }
```

У Kotlin можливо також створити змінну, що матиме функціональний тип.

```
val mutator = { item: String -> "-$item"  }
```

Оголошення параметрів відбувається всередині {}, а тіло – після знака ->. Надалі змінну можна використати як аргумент для функції вищого порядку, або ж викликати відразу.

```
val items = mutableListOf("a", "b", "c", "d", "e")
           //виклик функції вищого порядку з екземпляром
↪  функціонального типу
           mutate(items, mutator)
           println(items)
           //виклик екземпляру функціонального типу
↪  (двома способами)
           println(mutator("test"))
           println(mutator.invoke("test"))
//Вивід:
//[-a, -b, -c, -d, -e]
//-test
//-test
```

Також як аргумент для функціонального типу можна передати анонімну функцію:

```
val items = mutableListOf("a", "b", "c", "d", "e")
           mutate(items, fun (s: String) = "-$s" )
```

Існує ще декілька методів передавання аргументів, проте в контексті цього посібника їх не розглядатимуть. Якщо лямбда-функція має лише один параметр, під час її виклику його можна не вказувати, а використовувати автоматичний `it`.

```
val items = mutableListOf("a", "b", "c", "d", "e")
items.forEach { println(it) }
```

Для повернення значення із лямбда-функції, можна використати `return`, або значення останнього виразу.

```
val items = mutableListOf("a", "b", "c", "d", "e")
//повернення останнього виразу
```

```
items.find {
        val suitable = it == "a"
        suitable
}
//повернення через return
items.find {
        val target = it == "d"
        return@find target
}
//коротка форма
items.find { it == "b" }
```

Важливою властивістю лямбда-функції є те, що ми можемо оголосити змінну за її межами та без проблем змінювати всередині лямбди.

```
val items = mutableListOf("a", "b", "c", "d", "e")
//змінна count
var count = 0
items.forEach {
        println(it)
        //змінюємо значення змінної count всередині
↪   лямбди
           count++
           }
println("Total: $count")
```

У прикладі нижче можна отримати доступ до методів об'єкта-приймача.

```
val mutator: String.(String) -> String = { str -> "$this
↪  ${str.uppercase()}" }
```

Об'єктом який приймає в даному випадку є об'єкт типу `String`, звертатись до якого можна через ключове слово `this`. В цілях оптимізації невеликих лямбда-виразів, існує механізм вбудованих

функцій. Такі функції позначають як `inline`. Під час використання такого модифікатора лямбда-функцію буде інтерпретовано компілятором як звичайний послідовний код. Поглянемо на приклад стандартної вбудованої функції `forEach`:

```
@kotlin.internal.HidesMembers
public inline fun <T> Iterable<T>.forEach(action: (T) ->
↪  Unit): Unit {
           for (element in this) action(element)
}
```

У цьому випадку лямбда-функція `action` буде вбудованою та не викличе додаткових навантажень, інакше, на кожній ітерації буде створюватись новий об'єкт типу `Function`, що виконуватиме необхідні інструкції, описані в лямбді. Крім того, у вбудованих функціях ми можемо без проблем викликати `return`, що цілком логічно.

```
fun do(){
val items = mutableListOf("a", "b", "c", "d", "e")
           items.forEachIndexed { index, str ->
                   println(str)
                   if(index == 2)
                        //відбудеться вихід із
↪  зовнішньої функції do
                       return
            }
           //сюди ми не дойдемо
          println("End")
}
```

На противагу `inline` існує й `noinline`, який зазначає, що ця лямбда-функція не може бути вбудована. Це ключове слово можна використовувати під час оголошення окремих параметрів у `inline` функції.

```
inline fun test(inlineFun1: () -> Int, inlineFun2: () ->
↪ Int, noinline noInlineFun1: () -> Int) {}
```

1.3.3. Деструктуризація, varargs та spread-оператор

За допомогою механізму реструктуризації у Kotlin можна зручно розмежувати об'єкт на декілька змінних, які одразу можна використовувати. Наприклад, нехай деяка функція повертає об'єкт типу, що містить декілька змінних, це можна записати так:

```
fun main(args: Array<String>) {
        //деструктуризований об'єкт
        val (type, age) = getFood()
        println(type)
        println(age)
}
```

Функція, що повертає об'єкт:

```
fun getFood() = Food("fast", 10)

class Food(val type: String, val age: Int) {
        /*для деструктуризації необхідно оголосити
↪ функції з назвами component1...componentN, що
↪ повертають відповідні змінні*/
        operator fun component1() = type
        operator fun component2() = age
}
```

Якщо один із компонентів деструктуризованого об'єкта не використовують, його можна позначити як `_`.

```
val (_, age) = getFood()
```

Для передання різної кількості аргументів у функцію Kotlin надає ключове слово `vararg`. Зазвичай `vararg` розміщують як

останній параметр, у іншому випадку, передання інших аргументів, що йдуть після нього, потрібно виконувати поіменно.

```
fun printWords(vararg words: String){
        words.forEach {
                println(it)
        }
}

//Виклик функції з різною кількістю аргументів
printWords("One", "Two", "Three")
printWords("Four", "Five")
printWords("Six")
```

У випадку, якщо vararg не є останнім параметром, використовуємо іменований аргумент.

```
fun printWords(vararg words: String, prefix: String){
    words.forEach {
        println("$prefix $it")
    }
}

printWords("One", "Two", "Three", prefix = "Prefix1")
printWords("Four", "Five", prefix = "Prefix2")
printWords("Six", "Seven", prefix = "Prefix3")
```

Для того, щоб передати як vararg аргументу масив даних, потрібно використати spread-operator $*$, який розкладає масив у список змінних.

```
val items = arrayOf("a", "b", "c", "d")
printWords(*items)
```

1.4. Класи та об'єкти

Kotlin підтримує парадигму об'єктно-орієнтованого програмування та надає багато зручних інструментів та можливостей, для уникнення зайвих конструкцій – усі вони генеруюся компілятором автоматично, потрібно лише правильно їх використати. Крім того, існує декілька нових типів класів, що значно полегшують життя та код. Про всі ці особливості, а також базові речі йтиметься у цьому розділі.

Створення класу. Об'єкти.

Створити клас у Kotlin просто, для цього потрібно використати ключове слово `class`.

```
class Car
```

Конструктори у Kotlin є двох типів: первинний та вторинний. Первинний конструктор використовують в оголошенні класу, вторинні (додаткові) в тілі. Під час оголошення класу, ми можемо одразу визначити властивості у первинному конструкторі.

```
//первинний конструктор із властивостями
class Car constructor(val type: String = "Toyota", val
↪  speed: Double = 280.0) {
        //вторинний конструктор
        constructor(car: Car): this(car.type, car.speed)
}
```

Як бачимо із прикладу, в конструкторі також можна використовувати значення за замовчуванням. За допомогою ключового слова `this` та : посилаємось на первинний конструктор. Якщо первинний конструктор не містить ніяких модифікаторів чи анотацій – ключове слово `constructor` можна опустити, зробивши таке:

```
class Car (val type: String = "Toyota", val speed:
↪  Double = 280.0)
```

Для додаткової ініціалізації можна використати блок ініціалізації `init`, у якому можна ініціалізувати необхідні властивості та виконати додатковий код. Таких блоків може бути декілька.

```
class Car (val type: String = "Toyota", val speed:
↪  Double = 280.0) {
    constructor(car: Car): this(car.type, car.speed)
        init {
        println("Блок ініціалізації")
        }
}
```

Для створення об'єкта класу нам потрібно записати його назву та передати дані у конструктор.

```
//оскільки є значення  за замовчеванням
val car1 = Car()
//використання первинного конструктора
val car2 = Car("KIA", 285.0)
//використання вторинного конструктора
val car3 = Car(car2)

//вивід значень властивостей
println("Type: ${car1.type} Speed: ${car1.speed}")
```

Клас також може і не мати явно визначеного конструктора, в такому випадку під час створення об'єкта не передаємо жодних аргументів.

Kotlin також надає можливість створювати анонімні об'єкти. Такі об'єкти мають тип `Any`, якщо їх не наслідують від якогось конкретного:

```
//створення анонімного об'єкта
val anonimousObject1 = object {
        val someString = "Hello"
        }
```

```
//вивід значення властивості
println(anonimousObject1.someString)
```

```
//створення анонімного об'єкта від Car
val anonimousObject2 = object: Car() {
val someString = "Hello"
                val info = "Type: ${this.type} Speed:
↪  ${this.speed}"
        }
//вивід значення властивості
println(anonimousObject2.info)
```

Зверніть увагу на тип анонімного об'єкта, наприклад, повертаючи його як значення функції.

Ще одна цікава річ у Kotlin – допоміжні об'єкти. Такі об'єкти визначають за допомогою ключового слова `companion` (за необхідності, їм також можна дати назву). По суті це «Одинак»[1] (також відомий як: Singleton), тому він може реалізовувати інтерфейси тощо. Розглянемо детальніше:

```
open class Car (val type: String = "Toyota", val speed:
↪  Double = 280.0) {
//створення  допоміжного об'єкта (ім'я вказувати не
↪  обов'язково)
        companion object Comp {
                fun defaultSpeed() = 250.0
        }
}
```

Приклад використання

```
//Для доступу використовуємо ім'я класу
val defSpeed1 = Car.Comp.defaultSpeed()
//Ім'я можна доп. об'єкта можна не вказувати
```

[1] «Одинак» – це породжувальний патерн проєктування який гарантує, що клас має лише один екземпляр, та надає глобальну точку доступу до нього.

```
val defSpeed2 = Car.defaultSpeed()

//отримання допоміжного об'єкта
val comObj = Car
```

Однією, дуже ефективною з точки зору зменшення об'єму коду опцією є оголошення об'єкта. Це надає можливість використовувати «одинака» без написання зайвого, як у Java.

```
object EngineBase {
        fun getInfo() = "..."
}

//виклик функції оголошеного об'єкта
EngineBase.getInfo()
//подібно до статичного методу, проте це <<одинак>>
```

Як бачимо з прикладу, все нагадує створення класу, проте з ключовим словом `object`. Щоб отримати об'єкт, достатньо просто вказати назву.

```
//отримання доступу до оголошеного об'єкту в якості
↪  змінної
val base = EngineBase
```

1.4.4. Властивості класу. Наслідування. Модифікатори доступу

У прикладах попереднього підрозділу було наведено зразки оголошення властивостей класу. Крім `val` (незмінних властивостей), можна також використовувати `var` (їх можна модифікувати). Kotlin дає можливість перевизначити методи доступу(геттер та сеттер) до властивостей класу.

```
open class Car (val type: String = "Toyota") {

var parts: Int = 0
                //перевизначення геттера
                get() {
                println("Getting part count")
                return field
                }
                //перевизначення сеттера
                set(value) {
                println("Setting part count * 2")
                field = value * 2
                }
}

val car = Car()
println(car.parts)
car.parts = 10
println(car.parts)
```

Результат роботи прикладу:
Getting part count
0
Setting part count * 2
Getting part count
20

Властивості також можна позначати як `lateinit`. Робиться це для того, щоб уникнути примусової ініціалізації ненульової властивості (в деяких випадках) та сказати компілятору, що це буде зроблено обов'язково пізніше.

```
open class Car (val type: String = "Toyota") {
        var subCar: Car // помилка, необхідна
 ↪  ініціалізація
}
```

```
open class Car (val type: String = "Toyota") {
        lateinit var subCar: Car // помилки немає

        //ініціалізуємо пізніше
        fun initSubCar(car: Car){
                subCar = car
        }
}
```

Єдине, lateinit не можна використовувати для примітивних типів. Існує також можливість перевірити, чи ініціалізована `lateinit` властивість.

```
fun check(car: Car){
    car::subCar.isInitialized
}
```

У Java всі класи неявно наслідують клас `Object`. У Kotlin все аналогічно, проте усі класи неявно наслідують клас `Any`. У попередніх прикладах Ви могли побачити ключове слово `open`. Його застосовують для позначення, що цей клас можна наслідувати (за замовчуванням, класи є приватними і їх наслідувати не можна).

```
open class Vehicle(val type: String) {
        fun run() {
                println("Vehicle is running!")
        }
}
```

Ключове слово `open` також можна застосовувати до функцій та властивостей класу. Приклад наслідування:

```
class Car(type: String) : Vehicle(type)
```

Як бачимо із прикладу, якщо клас, від якого наслідуємо, має

основний конструктор, можна одразу його викликати під час оголошення. Зробити це можна так:

```
class Car : Vehicle {
        constructor(type: String): super(type)
}
```

Для того, щоб перевизначити функцію, потрібно, щоб вона була позначена `open` (або `abstract`), головно, використати ключове слово `override`:

```
open class Vehicle(val type: String) {
           //функція позначена open (її можна
↪  перевизначити)
           open fun run() {
                println("Vehicle is running!")
    }
}

class Car(type: String) : Vehicle(type){
        override fun run() {
                println("Core is running!")
    }
}
```

Також є можливість викликати функцію із класу, від якого наслідуємо за допомогою ключового слова `super`.

```
override fun run() {
        //виклик Vehicle#run
    super.run()
}
```

Можливо також указати конкретний супертип, від якого буде викликана функція.

```
super<Vehicle>.run()
```

Серед модифікаторів доступу у Kotlin є такі: `public`, `protected`, `private`, `internal`. Якщо модифікатора доступу немає то, за замовчуванням, використовуватиметься `public`. Розглянемо детальніше принцип роботи кожного модифікатора:

- `public` – видимість всюди; члени класу видимі усім, хто має доступ до цього класу;
- `protected` – лише для членів класу; члени класу видимі лише всередині класу та його спадкоємцях;
- `private` – видимість лише всередині файлу; члени класу видимі лише всередині класу та його внутрішніх членах (проте не навпаки);
- `internal` – видимість лише всередині модуля.

Крім того, оголосити змінну або функцію можна і на найвищому рівні:

```
package test

//видно всюди
val someValue = 10
//видно лише в рамках модуля
internal val someInternalValue = 25
//видно лише в середині цього файлу
private val onlyHereValue = 8
```

Приклад застосування модифікаторів доступу:

```
open class Vehicle(val type: String) {
        //private властивість
           private val somePrivateValue = 0
           //protected властивість
           protected val someProtectedValue = 0
```

```
            //internal властивість
            internal val someInternalValue = 0

        open fun run() {
                println("Vehicle is running!")
        }
}

class Car(type: String) : Vehicle(type){
        override fun run() {
                //помилка, властивість доступна лише в
↪   межах класу Vehicle
            this.somePrivateValue
            //можна використовувати, оскільки Car
↪   наслідує Vehicle            this.someProtectedValue
                   //можна використовувати, оскільки все
↪   в одному модулі
                   this.someInternalValue
        }
}
```

1.4.4. Інтерфейси та абстрактні класи. Data-класи. Sealed-класи. Enum-класи

Створити інтерфейс у Kotlin можливо за допомогою ключового слова `interface`.

```
interface IVehicle {
        var owner: String
        fun getType(): String
        fun getSpeed(): Double
        fun run(){
                println("Vehicle is running!")
        }
}
```

Як бачимо з прикладу, в інтерфейсі можуть міститися функції як без реалізації та і з нею (відповідно, такі функції можна не перевизначати), а також властивості. Реалізуємо на прикладі наведений вище інтерфейс `IVehicle`:

```
open class Car: IVehicle  {
        override var owner = "Peter"
        override fun getType(): String {
                TODO("Not yet implemented")
        }
        override fun getSpeed(): Double {
                TODO("Not yet implemented")
        }
}
```

Інтерфейси також можуть наслідувати інші інтерфейси, добавляючи щось нове, або модифікуючи наявне.

```
interface ICar: IVehicle {
    fun getModel(): String
        override fun run() {
                println("Car is running!")
        }
}
```

Крім інтерфейсів, поговоримо про особливий тип класів – абстрактні класи. Абстрактні класи оголошують за допомогою ключового слова `abstract`. Все, що позначене цим, не може мати своєї реалізації, її потрібно створити у нащадках. За цією логікою, абстрактний клас можна не позначати як `open`.

```
//абстрактний клас
abstract class Vehicle {
//абстрактні функції
        abstract fun getSpeed(): Double
```

```
        abstract fun getPosition(): Pair<Double, Double>
}
```

Створимо клас `Car`, що наслідує абстрактний клас `Vehicle`:

```
class Car: Vehicle() {
        //необхідно реалізувати абстрактні функції
        override fun getSpeed(): Double {
                TODO("Not yet implemented")
        }
        override fun getPosition(): Pair<Double, Double>
 ↪  {
                TODO("Not yet implemented")
        }
}
```

Data-класи, або класи даних у Kotlin є досить поширеним типом класів, які використовують для зберігання даних. Під час створення data-класу, компілятор автоматично генеруватиме функцію `toString`, `hashCode`, `equals`, функції-компоненти (які можна використати для деструктуризації) та функцію `copy`. Варто наголосити, що під автогенерацію підпадають лише ті властивості, що оголошені в тілі основного конструктора, звідси також випливає, що конструктор повинен мати хоча б одну властивість.

```
data class DataEntry(val id: Int, var description:
 ↪  String)
```

Як бачимо із цього прикладу, для оголошення data-класу використовують ключове слово `data`. Також усі властивості повинні бути помічені як `val` або `var`. Крім деструктуризації, у data-класів існує метод `copy`, що дає можливість створити новий об'єкт, скопіювавши лише властивості, перелічені у конструкторі.

Для класів та інтерфейсів можна застосувати модифікатор `sealed`. Sealed-класи у Kotlin мають досить цікаву функцію. Наслідувати від таких класів можна лише до процесу компіляції.

Отже, після компіляції ніхто не зможе вже реалізувати `sealed`-інтерфейс або розширити sealed-клас. Оголосити клас як `sealed` можна за допомогою однойменного ключового слова:

```
sealed class ISomeBehaviour
```

Наведений вище sealed-інтерфейс можна реалізувати лише в межах його пакета та модуля.

Також `sealed`-типи можна зручно використовувати у конструкції `when`, подібно до всіх випадків `enum`-класу. Якщо ми знаємо спадкоємців наперед, та у конструкції `when` їх урахували, тоді нам не потрібно використовувати `else` умову.

Останнім типом у цьому підрозділі розглянемо перелічувальні `enum`-класи. Якщо коротко – такий тип класу дозволяє оголошувати константи-об'єкти. Такі класи можуть реалізувати інтерфейси, функції тощо:

```
enum class Vehicle(val info: String): IVehicle {
          CAR("Car type"), MOTOR("Motorcycle type"),
 ↪  CYCLE("Bicycle type")
}

interface IVehicle
```

Розглянемо також приклад виклику деяких синтетичних методів, що наявні в усіх `enum`-класах:

```
//використання методу valueOf
val car = Vehicle.valueOf("CAR")
println(car.info)
//використання методу values
println(Vehicle.values().map { it.name.toLowerCase() })
//ім'я константи
println(car.name)
//порядковий номер константи
println(car.ordinal)
```

1.4.4. Вкладені та внутрішні класи

У Kotlin можна без проблем оголосити один клас/інтерфейс усередині іншого класу/інтерфейсу.

```
//зовнішній клас
class Vehicle(val name: String){
    val info = "some info"
    //внутрішній клас
        class Properties {
            fun verify(){
                        info //доступу немає
                }
        }
}

//створення об'єкта типу Properties
val props = Vehicle.Properties()
```

Як показано у прикладі, доступу до властивості зовнішнього класу `Vehicle` у внутрішньому класі `Properties` немає. Щоб виправити це, можна використати інший тип класу – вкладений, або `inner`.

```
//зовнішній клас
class Vehicle(val name: String){
        val info = "some info"
        //вкладений клас
        inner class Properties {
                fun verify(){
                info //доступ є
                }
        }
}

//створення об'єкта типу Properties
val props = Vehicle("car").Properties()
```

1.4.4. Узагальнення

Типізовані параметри (Generics) у Kotlin у простому варіанті використовують майже аналогічно до Java. Розглянемо приклад створення класу з типізованим параметром:

```
class Car(val speed: Double)

class Vehicle<T>(val type: T) {
                    fun getVehicle() = type
}

//Тип визначається автоматично, також його можна вказати
↪  явно у <>
val carType = Vehicle<Car>(Car(10.0))
println(carType.getVehicle().speed)
```

Як бачимо з прикладу, замість `T` ми можемо підставити будь-який тип та згодом з ним працювати. Ви стикнетесь із цим, створюючи колекції та вказуючи їх тип.

```
val list = mutableListOf<String>()
```

Із функціями подібна ситуація.

```
interface IVehicle
fun <T : IVehicle> runVehicle(vehicle: T) {...}
```

У цьому прикладі наведено створення функції з типізованим параметром. Зверніть увагу на `<T : IVehicle>`, таким способом ми зазначаємо, що функція прийматиме тільки реалізації інтерфейсу `IVehicle`. Також можливо визначити декілька таких залежностей за допомогою ключового слова `where`:

```
interface IVehicle
interface IPlane
```

```
fun <T> runVehicle(vehicle: T) where T : IPlane, T :
↪  IVehicle {
        ...
}
```

Також можливо визначити типізований параметр як `out`. Отже, вважатимемо, що такий параметр буде лише повертатись методами класу.

```
class Vehicle<out T>(val type: T) {
    fun getVehicle() = type
}
```

На противагу йому існує також позначення `in`, орієнтоване на прийом узагальненого типу.

```
class Vehicle<in T> {
        fun getInfo(vehicle: T){ }
}
```

1.4.4. Делегати та делеговані властивості. Лінива ініціалізація.

За допомогою делегатів Kotlin дає змогу значно скоротити код, використовуючи гнучкий механізм передання об'єкта, якому будуть делеговані усі функції класу (тільки публічні), у який він передається. Розглянемо приклад:

```
fun main(args: Array<String>) {
        //створюємо об'єкт типу Vehicle
        val vehicle = Vehicle()
        //створюємо об'єкт типу Car передаючи в нього
↪  vehicle
        val car = Car(vehicle)
        //буде виведено: Vehicle is running
        car.run()
```

```
}

interface IVehicle {
        fun run()
}

class Vehicle: IVehicle {
        override fun run() {
                println("Vehicle is running")
        }
}

class Car(vehicle: Vehicle): IVehicle by vehicle
```

Як бачимо з прикладу, клас `Car` делегує реалізацію своїх функцій об'єкту класу `Vehicle`. Зверніть увагу на ключове слово `by`, за допомогою якого це відбувається. Отже, нам не потрібно реалізовувати функції інтерфейсу `IVehicle` знову. Звісно, все, що потрібно, можна перевизначити за допомогою ключового слова `override` та задати свою поведінку.

Делегати також застосовують і до властивостей. Наприклад, ми можемо створити власний делегат, якому будуть делеговані методи `get`- та `set`-властивості у відповідні делегату методи `getValue` та `setValue`.

```
class Vehicle {
        var speed by SpeedDelegate()
}

class SpeedDelegate {
        var value = 0.5

        operator fun getValue(obj: Vehicle, prop:
 ↪  KProperty<*>): Double {
        println("get double value from ${prop.name} as
 ↪  ${this.value}")
```

```
                return this.value
}

operator fun setValue(obj: Vehicle, prop: KProperty<*>,
↪  data: Double) {
        println("set double value to $data")
        this.value = data
        }
}
```

Для роботи достатньо створити у класі делегата функції

```
operator fun getValue(obj: Vehicle, prop: KProperty<*>)
```

що першим параметром приймає тип класу, в якому є властивість, та повертає тип самої властивості. Аналогічно і з operator

```
fun setValue(obj: Vehicle, prop: KProperty<*>, data:
↪  Double)
```

`data` – присвоюване значення. Запустимо приклад на виконання:

```
val vehicle = Vehicle()
vehicle.speed
vehicle.speed = 10.0
```

Вивід:

get double value from speed as 0.5

set double value to 10.0

Якщо ж властивість є незмінною `val`, ми можемо не реалізовувати функцію setValue всередині делегата.

Одним із поширених делегатів є `lazy()`, або лінива ініціалізація. Використання цього делегата означає, що весь розміщений у переданій йому лямбді код виконується лише один раз за першим зверненням до нього, усі наступні рази ми отримуватимемо одне і теж саме значення.

```
fun main(args: Array<String>) {
        //виводимо значення перший раз
        println(lazySample)
        //очікуємо 5 секунд
        Thread.sleep(5000)
        //повторний виклик дасть те саме значення
        println(lazySample)
}

val lazySample by lazy {
        System.currentTimeMillis()
}
```

Вивід:
1663638187050
1663638187050
Як і очікували, отримуємо однакові значення часу.

Ще одним корисним підходом є використання `Delegates.observable`. Ця властивість приймає ініціалізуюче значення та лямбду, що містить опис властивості, її старе та нове значення.

```
        observableSample = 5
        observableSample = 12

var observableSample by Delegates.observable(10) {
        prop, oldValue, newValue -> println("Old value:
↪  $oldValue, new value: $newValue")
}
```

Запустивши цей приклад, отримаємо:
Old value: 10, new value: 5
Old value: 5, new value: 12

Якщо нам потрібно перевірити умову перед присвоєнням значення властивості, можемо використати делегат `Delegates.vetoable`.

```
        observableSample = 5
        observableSample = 12
        observableSample = 15

var observableSample by Delegates.vetoable(10) {
        prop, oldValue, newValue -> println("Old value:
↪  $oldValue, new value: $newValue") newValue > 10
}
```

З результату видно, що значення **5** не було присвоєно властивості, оскільки `vetoable` повертає умову, що нове значення має бути більше **10**:

Old value: 10, new value: **5**
Old value: **10**, new value: 12
Old value: 12, new value: 15

Використання делегованих властивостей надає можливість передавати властивості за допомогою асоціативного списку.

```
fun main(args: Array<String>) {
        val vehicle = Vehicle(mapOf("type" to 0, "count"
↪  to 5))
}

class Vehicle(params: Map<String, Int>){
        val type by params
        val count by params
}
```

Із наведеного прикладу можна інтуїтивно здогадатися, що в як ключ у мапі виступає назва властивості, а в як значення – значення відповідної властивості.

Якщо ми не маємо інформації про тип, використовуємо $*$.

```
fun something(list: List<*>){...}
```

Такий знак називають star-projection.

1.5. Інші концепти

У цьому розділі посібника ми опишемо деякі важливі та корисні можливості Kotlin, для написання безпечного та ефективного коду.

1.5.5. Null-безпека та приведення типів

Kotlin, на відміну від Java, чітко розмежовує посилання, що можуть мати значення `null` від посилань, що такого значення ніколи не матимуть.

```
var data: Int = 5
data = null          //помилка, data має тип Int, якому
↪  не можна присвоїти null-значення
```

Під час спроби встановити значення змінної як `null` отримаємо помилку. Це пов'язане з тим, що ми явно визначили тип змінної як такий, що не може містити `null`-значення. Виправимо цю ситуацію:

```
var data: Int? = 5
data = null                 //помилки немає
```

З прикладу вищу видно, що тепер `null`-значення присвоєно. Для цього потрібно додати до типу суфікс `?`. Якщо ж ми спробуємо викликати якийсь метод, отримаємо помилку:

```
var data: Int? = 5
data = null                 //помилки немає
data.digitToChar()          //помилка, небезпечний виклик
```

Ми не можемо напряму викликати метод для змінної, що може мати `null`-значення. Для таких випадків існує так званий безпечний виклик (safe-call).

```
var data: Int? = 5
...
data?.digitToChar()  //метод буде викликаний лише тоді,
↪  коли data не буде рівна null
```

Якщо data матиме null-значення, метод digitToChar не буде викликано, та повернено null. За допомогою такого методу можна створювати ланцюги безпечних викликів, та якщо одна з ланок поверне null – ланцюг буде перервано.

Для перевірки, чи є значення null, можна використати стандартну перевірку типу if(data != null), або використати так званий елвіс-оператор ?:.

```
var data: Int? = 5
val res1 = data ?: -1        //еквівалент if(data !=
↪  null) res1 = data else res1 = -1
val res2 = data?.digitToChar() ?:
↪  '0'        //еквівалент if(data != null) res2 =
↪  data.digitToChar() else res2 = '0'
```

Наостанок ми можемо викликати метод із null-змінної, явно вказавши компілятору, що ми впевнені у своїх діях за допомогою !!.

```
var data: Int? = 5
              ...
data!!.digitToChar()  //помилки немає, проте є ризик
↪  виникнення NullPointerException
```

Поговоримо тепер про приведення типів. Для перевірки, чи належить/не належить змінна відповідному типу, використовують оператор is та !is, відповідно.

```
val data: Any = "text"
println(data is String)                //true
println(data !is String)        //false
println(data is Int)                //false
```

Kotlin вміє також автоматично визначати тип змінної після вдалої перевірки на відповідність певному типу.

```
val data: Any = "text"
if(data is String){
        //автоматично визначено тип як String
        println(data.substring(2))
        }
        //також в конструкції when
when(data){
        is String -> println(data.substring(2))
    }
```

Привести змінну до відповідного типу можна за допомогою оператора приведення as та безпечного приведення типу as?

```
val data: Any = "text"

//небезпечне приведення (отримаємо помилку в ході
↪  виконання коду)
val value1 = data as Int
//безпечне приведення (отримаємо null у випадку помилки
↪  приведення)
val value2 = data as? Int
```

1.5.5. Порівняння в Kotlin

У Kotlin порівняти об'єкти можна декількома методами. Перший із них – це перевірка рівності структури об'єктів (`==` та `!=`).

```
var data1: Any = "text1"
var data2: Any = "text2"

println(data1 == data2)     //false (значення різне)
```

```
data2 = "text1"
println(data1 == data2)     //true (значення однакове)
```

Додатково можна визначити, як структурно порівнюватимуться об’єкти на свій лад. Для цього потрібно перевизначити функцію `equals`.

Другий – перевірка на посилання (`=== та !==`). Якщо дві змінні вказують на один і той самий об’єкт, ми можемо говорити про їхню рівність.

```
var n1: Any = 1500
var n2: Any = 1500

println(n1 == n2)               //true (структурна
↪  рівність)
println(n1 === n2)              //false (різні об'єкти)
n2 = n1
println(n1 === n2)              //true        (тепер
↪  об'єкт однаковий)
```

1.5.5. Ключове слово this

Ключове слово `this` використовують для посилання на об’єкт класу, в якому його викликають. Наприклад, для доступу до членів. Якщо ж є необхідність звернутись до члена із іншої області, можна використати неявні мітки.

```
class SomeClass {
        val par = 1
        inner class SomeInnerClass {
                val par = 2
                fun show(){
                //звернення до змінної
↪  SomeInnerClass#par
                println(this.par)
```

```
                //звернення до змінної SomeClass#par
↪  (@SomeClass - неявна мітка)
                println(this@SomeClass.par)
                }
        }
}
```

this також можна використовувати у функціях-розширеннях як об'єкт-приймач.

```
"some text".run {
        println(this)
}
```

1.5.5. Перевизначення операторів

У Kotlin існує можливість перевизначити унарні, бінарні, інкремент/декремент, тощо оператори для свого класу. Операторів визначають наявністю ключового слова operator біля функції-члена класу, або функції-розширення та мають певну сигнатуру та назву, притаманні функції, що потрібно перевизначати. Розглянемо приклад, у якому у нас є клас, котрому буде перевизначено арифметичний +. Таким способом ми зможемо перетворити два об'єкти цього класу в третій, що буде сумою з визначеною нами логікою.

```
fun main(args: Array<String>) {
        //створюємо два об'єкти
        val part1 = Part(5.0)
        val part2 = Part(10.0)
        //сумуємо два об'єкти викоритосвуючи оператор +,
↪  що був перевизначений
        val part3 = part1 + part2
        println(part3.size)         //15.0
}
```

```
class Part(val size: Double){
        //перевизначаємо оператор + (plus) та формуємо
↪  клас-суму з властивістю розміру size
        operator fun plus(otherPart: Part): Part =
↪  Part(this.size + otherPart.size)
}
```

1.5.5. Функції-розширення

Однією із цікавих й водночас дуже зручних можливостей Kotlin є функції-розширення. За допомогою них ми можемо "розширювати" класи, додаючи до них новий функціонал без зайвих потуг. Розрізняють функції-розширення, властивості-розширення та розширення для допоміжних об'єктів. Розглянемо приклад із функцією-розширенням:

```
fun main(args: Array<String>) {
        val text = "Hello"
        //виклик функції addPrefix як члена класу String
        println(text.addPrefix("-"))
}

//функція-розширення, де this вказує на поточний об'єкт,
↪  з яким іде робота
fun String.addPrefix(prefix: String) = "$prefix$this"
```

Як бачимо із прикладу, після додавання функції-розширення для класу `String` ми викликаємо функцію-розширення як член класу `String` без його модифікації. Зазвичай розширення оголошують на рівні пакетів.

1.5.5. Функції області видимості

Розглянемо кілька корисних функцій, які можна викликати для певного об'єкта. За допомогою цих функцій можна виконати

певний код завдяки лямбда-виразу над об'єктом (називатимемо його контекстом), для якого ця функція була викликана. Існує декілька видів функцій області видимості, кожен з яких варто застосовувати за певних потреб.

Функція `run` дає можливість отримати доступ до об'єкта за допомогою контексту `this`, повертає результат виконання лямбда-виразу та використовується для налаштування об'єкта та повернення результату. Крім того, `run` можна застосовувати в комбінації з функцією `let` для виконання дій у разі повернення `null`-значення, або виконання коду без контекстного об'єкта. Розглянемо три випадки на прикладі нижче:

```
//виконання блоку коду без контекстного об'єкта
val result = run {
        val a = 5
        val b = 10
        Pair(a, b)
}
println(result)

//виконання блоку коду над об'єктом
val vehicle = Vehicle()
val info = vehicle.run {
        this.speed = 20.0
        val data = this.parts.joinToString("[", " - ",
 ↪  "]")
        Pair(this.speed, data)
}
println(info)

//комбінація з let
val data: String? = null
//код ...
//у разі, якщо data = null, викликаємо блок run
data?.let { println(it) } ?: run { println("No data
 ↪  found") }
```

Функція let дає змогу безпечно викликати блок коду для значення, що може бути null'ом. Також за допомогою let можна викликати декілька функцій перед поверненням результату виконання лямбда-виразу. Контекстний об'єкт доступний по замовчуванню як it, проте його можна назвати як завгодно.

```
//let
val data: String? = null
//у разі, якщо data = null - код лямбди не виконається
val length = data?.let { str ->
        println(str)
        str.length
}
```

Функція with отримує об'єкт як аргумент та повертає результат виконання лямбда-виразу. Зазвичай використовують для виклику послідовності функцій-роботи з об'єктом, не повертаючи певного результату. Крім того, за допомогою неї можна зручно змінювати властивості об'єкта.

```
val data = "Some text"
with(data){
        println("Length: $length")
        println("Capitalized: ${capitalize()}")
        //etc
}

val vehicle = Vehicle()
with(vehicle){
        speed = 25.0
        parts.add("Wheel")
}
```

Функція apply діє інакше – вона повертає сам контекстний об'єкт та використовується, головно, для налаштування властивостей об'єкта. Контекстний об'єкт доступний як this. Нижче наведено класичний приклад використання apply:

```
val vehicle = Vehicle().apply{
        speed = 25.0
        parts.add("Wheel")
}
```

Функція `also` також повертає контекстний об'єкт, проте всередині лямбди він доступний аргументу `it`, аналогічно `let`. Використовують `also` зазвичай у ситуаціях, коли ми не застосовуємо або не змінюємо контекстний об'єкт, проте нам потрібно отримати його для виконання подальших дій.

```
val vehicle = Vehicle().apply{
        speed = 25.0
        parts.add("Wheel")
        }.also {
                println("Speed = ${it.speed}")
                println("Parts = ${it.parts}")
                }.runAndGet()
println(vehicle.speed)
```

1.6. Колекції в Kotlin

У межах цього розділу розглянемо основні типи колекцій у Kotlin, їх застосування та зручні та ефективні методи роботи із ними. Основними типами колекцій у Kotlin є `List`, `Set` та `Map`. `List` та `Set` є спадкоємцями від інтерфейсу `Collection`, який надає методи для реалізації простої незмінної колекції.

1.6.6. Загальний огляд. Ітератори. Діапазони. Послідовності

`List` – простий список елементів, розташованих у тому порядку, в котрому вони додавались в колекцію. Елементи у такому списку можуть повторюватись, також є можливість звернення до них по індексу. Крім того, можуть містити `null`-значення.

```
//змінювати такий список не можна
val immutableList = listOf("a", "b", "c", "d", "e")
println(immutableList)
immutableList.add("f")              //помилка
println(immutableList.get(0))
println(immutableList[2])

//список з можливістю модифікації елементів
val mutableList = mutableListOf("a", "b", "c", "d", "e")
mutableList.add("f")
println(mutableList)
mutableList.removeAt(0)
println(mutableList)
```

`Set` – список, який зберігає лише унікальні елементи, в тім числі і `null`.

```
//порядок зберігається
val immutableSet = setOf("a", "b", "c", "d", "e")
println(immutableSet)

val mutableSet = mutableSetOf("a", "b", "c", "d", "e")
mutableSet.add("f")
println(mutableSet)
mutableSet.remove("a")
println(mutableSet)

//порядок не зберігається
val mutableHashSet = HashSet<String>()
mutableHashSet.addAll(immutableSet)
println(mutableHashSet)
```

На відміну від `List` та `Set`, `Map` зберігає елементи у вигляді ключ-значення та не є спадкоємцем `Collection`. Як можна здогадатися, ключі для наборів значень є унікальними.

```
//незмінювана мапа, у якій ключ типу String, а значення
↪  - Int
val immutableMap = mapOf("User 1" to 2, "User 2" to 5,
↪  "User 3" to 10)
println(immutableMap)
//список ключів та значень
println(immutableMap.keys)
println(immutableMap.values)
//список наборів даних ключ-значення
println(immutableMap.values)
//використання в циклі for
for((key, value) in immutableMap)
        println("Key = $key Value = $value")
//ітерація
immutableMap.forEach{ (key, value) ->
        println("Key = $key Value = $value")
}
//доступ по ключу
println(immutableMap["User 1"])
//перевірка наявності ключа
if("User 3" in immutableMap)
        println("Found user 3")
//альтернативний варіант перевірки наявності ключа
if(immutableMap.containsKey("User 5"))
        println("Found user 5")
//перевірка наявності значення
if(5 in immutableMap.values)
        println("Found point 5")
```

Для порівняння двох `List` достатньо, щоб вони були однакового розміру, мали однаковий порядок елементів та щоб всі елементи були структурно однаковими. Для `Set` достатньо, щоб два списки мали однаковий розмір та однакові елементи у іншому списку. `Map` також уважають рівними, якщо вони мають однакові пари ключ-значення.

Розглянемо ще декілька прикладів для створення та роботою

із колекціями та мапами:

```
//створюємо список та ініціалізуємо елементи значеннями
↪  на 1 більше, ніж індекс
val array = List(5) { index -> index + 1}
println(array)

val vehicleMap = mutableMapOf<String, Vehicle>()
//додавання пари ключ-значення у мапу
vehicleMap["Opel"] = Vehicle(280.0)

//створення хеш-сету за допомогою його конструктора
val hashSet = HashSet<String>(50)

//перетворення immutable List у mutable (створюється
↪  незалежна копія)
val immutableList = listOf<String>()
val mutableList = immutableList.toMutableList()

//сумування двох списків
val list1 = listOf("a", "b")
val list2 = listOf("c", "d")
val sumList = list1 + list2
println(sumList)
```

Крім ітерації колекцій через `for`, Kotlin надає такий інструмент, як ітератор (`Iterable<T>`). У ітераторів доступні методи для перевірки наявності у них наступного (або попереднього) елементу, видалення елемента, заміна тощо. Розглянемо приклади застосування різних ітераторів:

```
//ітератор для незмінного списку
val immutableElements = listOf("a", "b", "c", "d", "e")
val immutableIterator = immutableElements.iterator()
while(immutableIterator.hasNext()) {
        val element = immutableIterator.next()
        println(element)
```

```
}
//ітератор для mutable списку, можна вилучати елементи
val mutableElements = mutableListOf("a", "b", "c", "d",
↪  "e")
val mutableIterator = mutableElements.iterator()
while(mutableIterator.hasNext()) {
        val element = mutableIterator.next()
        if(element == "c")
                mutableIterator.remove()
}
//forEach
mutableElements.forEach {   element ->
        println(element)
}
//forEach з індексацією
mutableElements.forEachIndexed{ index, element ->
println("$index: $element")
}
//ітератор списку, рух вперед та назад
val listIterator = immutableElements.listIterator()
while (listIterator.hasNext()){
        val next = listIterator.next()
        println(next)
}
while(listIterator.hasPrevious()){
val prev = listIterator.previous()
println(prev)
}
//ітератор для mutable списку, заміна та додавання
↪  елементів під час ітерації
val mutableListIterator = mutableElements.listIterator()
while (mutableListIterator.hasNext()){
        val next = mutableListIterator.next()
        if(next == "a")
                mutableListIterator.set("-")
                else mutableListIterator.add("+")
```

```
        }
println(mutableElements)
```

У попередніх розділах ми ознайомились з діапазонами – певним інтервалом чисел. Діапазони (а також прогресії) найчастіше використовують у `for` та під час перевірки належності числа інтервалу.

```
val diap = 0..10    //IntRange - діапазон цілих чисел
↪  0-10 включно
//ітерація в межах діапазону
for (i in diap)
        println(i)
//перевірка входження в діапазон
if(8 in diap)
        println("8 in 0-10")
//перевірка не входження в діапазон
if(15 !in diap)
        println("15 not in 0-10")
//діапазон від 0 до 9
val exl = 0 until 10
//прогресія з кроком 2
val step = 0..100 step 2
//ітерація з кроком
for (i in step)
        println(i)
//ітерація по прогресії від 30 до 0 з кроком 5
for(i in 30 downTo 0 step 5)
        println(i)
```

Як бачимо з прикладу, діапазони представлені класами `IntRange`, `LongRange` тощо, прогресії – `IntProgression`, `LongProgression` тощо.

Як альтернативу `Iterable<T>` у Kotlin також є ще одна цікава річ – `Sequence<T>`, або послідовність. Відрізняється послідовність тим, що під час обробки елементів, кожний крок відбувається

один за одним для кожного елемента, водночас як `Iterable<T>` спочатку виконує певну дію над усією колекцією, потім створює нову колекцію із результатами та переходить до наступного кроку. Така поведінка дає можливість у деяких випадках значно скоротити кількість кроків та підвищити швидкість роботи. Також у послідовності задіяне ліниве опрацювання результатів, що під час застосування з відносно невеликими або простими даними може, навпаки, погіршити ситуацію.

Подивимось на приклад, як працює `Iterable<T>` та `Sequence<T>`:

```
val iterable = listOf("Apple", "Banana", "Orange",
↪  "Cucumber")
iterable.filter {
        println("filter $it")
        it.length > 4
}.map {
        println("map $it")
        "-$it"
}.forEach {
        println("each $it")
}

//створення послідовності  за допомогою
↪  функції-розширення asSequence
val sequence = iterable.asSequence()
sequence.filter {
        println("filter $it")
        it.length > 4
}.map {
        println("map $it")
        "-$it"
}.forEach {
        println("each $it")
}
```

Запустимо на виконання та отримаємо:

Для `Iterable<T>`:	Для `Sequence<T>`:
filter Apple	filter Apple
filter Banana	map Apple
filter Orange	each -Apple
filter Cucumber	filter Banana
map Apple	map Banana
map Banana	each -Banana
map Orange	filter Orange
map Cucumber	map Orange
each -Apple	each -Orange
each -Banana	filter Cucumber
each -Orange	map Cucumber
each -Cucumber	each -Cucumber

1.6.6. Масиви, списки

У Kotlin масиви реалізовано класом `Array<T>`. Створити масив можна за допомогою функцій `arrayOf`, `arrayOfNulls`, `emptyArray`. Для роботи з масивами Kotlin також надає функції-розширення, наприклад, `plus` для додавання нового елемента. В такому разі буде повернено новий масив. Розглянемо, для прикладу, створення масивів:

```
//масив
val array = arrayOf("a", "b", "c", "d")
//додавання елементу за допомогою функції plus (новий
↪  масив)
val newArray = array.plus("e")
println(newArray.size)

//масив типу Int
val intArray = intArrayOf(1, 2, 3, 4)
//масив, заповнений null-значеннями
val nullArray = arrayOfNulls<String>(10)
```

```
//порожній масив
val emptyArray = emptyArray<String>()
```

Зверніть увагу, для масивів стандартних типів існують класи `IntArray`, `LongArray` тощо. Наприклад, еквівалентом `IntArray` у Java є `int []`, тож можна використовувати і їх.

```
//створення масиву із 5 елементів (усі елементи матимуть
↪  значення 0)
val intArray1 = IntArray(5)
/створення масиву із 5 елементів (усі елементи матимуть
↪  значення 10)
val intArray2 = IntArray(5) { index -> 10 }
```

Масив також можна перетворити у список за допомогою функції `toList`/`toMutableList`.

```
//перетворення масиву у список MutableList
val list = array.toMutableList()

//створення списку типу Int
val listOfInts1 = mutableListOf<Int>()
//створення списку типу Int з можливістю null-значень
val listOfInts2 = mutableListOf<Int?>()
//створення незмінного списку типу Int
val immutableList = listOf<Int>()
```

Доступ до елементів масиву та списку відбувається класично.

```
//доступ до першого елементу масиву
array[0]
//доступ до першого елементу списку
list[0]
```

1.6.6. Основні операції: перетворення, фільтрація, вибірка, групування

Kotlin містить безліч зручних та ефективних наборів інструментів для роботи із колекціями, починаючи від різноманітних перетворень, завершуючи сортуванням та об'єднанням. Розглянемо приклади використання найпоширеніших операцій над колекціями, які можна використати за подальшої роботи із даними.

Для перетворення елементів колекції використовують метод `map`. Цей метод дає можливість виконати лямбда-функцію над кожним елементом колекції та повернути результат у вигляді нової колекції.

```
val list = listOf("Hi", "my name is", "how are you",
↪  "great")
//перетворення колекції String в колекцію Int, що
↪  містить довжину рядків
val transformed = list.map { element -> element.length }
println(transformed)    //[2, 10, 11, 5]

//для роботи з null-значеннями (поверне тільки не
↪  null-значення)
val notNulls = list.mapNotNull {
        element ->
val length = element.length
        if(length < 2) null else length }

//з індексацією
val indexed = list.mapIndexed { index, element -> index
↪  * element.length }

//для Map
val map = mapOf<String, Double>("Opel" to 280.0, "Reno"
↪  to 270.0)
//перетворення для ключів
val mappedKeys = map.mapKeys { entry -> entry.key.length
↪  }
```

```
//перетворення для значень
val mappedValues = map.mapValues { entry -> "Speed:
↪ ${entry.value}"}
//перетворення у список Vehicle
val mappedVehicle = map.map { entry ->
↪ Vehicle(entry.key, entry.value) }
```

Для групування елементів двох списків можна використати функцію `zip`, а для протилежної операції `unzip`.

```
val list1 = listOf("a", "b", "c", "d")
val list2 = listOf(0, 1, 2, 3)

val zipResult = list1.zip(list2)
println(zipResult)  //[(a, 0), (b, 1), (c, 2), (d, 3)]

//протилежна zip операція:
println(zipResult.unzip())  //([a, b, c, d], [0, 1, 2,
↪ 3])
```

Із елементів списку за допомогою функції `associate`, `associateWith` та `associateBy` можна створити асоціативний список (мапу `Map`).

```
val list = listOf("a", "b", "c", "d")
val associate = list.associate { element ->
↪ Pair(element.toUpperCase(), element.length)}
//на виході отримуємо мапу ключ-значення: {A=1, B=1,
↪ C=1, D=1}
println(associate)

//елементи списку є ключами, змінюємо лише значення:
↪ {a=1, b=1, c=1, d=1}
val associateWith = list.associateWith { element ->
↪ element.length}
println(associateWith)
```

```
//елементи списку є значеннями, змінюємо лише ключі:
↪ {1=d} (тому що довжина всіх елементів 1, тому ключ
↪ лише один)
val associateBy = list.associateBy { element ->
↪ element.length }
println(associateBy)
```

Коли у нас є список, що містить у собі декілька інших списків, об'єднати усі елементи в один результуючий список можна за допомогою функції `flatten`.

```
val list = listOf(listOf("a", "b", "c"), listOf("d",
↪ "e", "f"))
//отримаємо один список [a, b, c, d, e, f]
println(list.flatten())
```

Для більш складних укладених конструкцій можна використати функцію `flatMap`.

```
val list = listOf(Vehicle(listOf("part1", "part2")),
↪ Vehicle(listOf("part1", "part2")))
//отримаємо один список об'єднаний по властивості parts:
↪ [part1, part2, part1, part2]
println(list.flatMap { it.parts })
```

Одним зі зручних інструментів для роботи зі списками є фільтрація, що відбувається за допомогою функції `filter` та декількох її модифікацій. За допомогою цієї функції можна назвати умову, стосовно якої відбуватиметься фільтрація та отримати результат у новому списку. Розглянемо приклади фільтрації:

```
//список типу Vehicle
val list = listOf(
        Vehicle(listOf("part1", "part2", "part3")),
        Vehicle(listOf("part1", "part2", "part3",
↪ "part4")),
```

```
        Vehicle(listOf()),
)

//отримуємо лише елементи, кількість частин в яких
↪  більше 3
val f1 = list.filter { it.parts.size > 3 }
println(f1)
//отримуємо лише елементи, що не задовольняють заданій
↪  умові
val f2 = list.filterNot { it.parts.isEmpty() }
println(f2)
//фільтрація з індексом
val f3 = list.filterIndexed { index, element -> index >
↪  0 && element.parts.isNotEmpty() }
println(f3)

//створимо список типу Any
val listOfAny = listOf("d", 0.5, 2f, null)
//отримуємо елементи лише типу String
val f4 = listOfAny.filterIsInstance<String>()
println(f4)
//отримуємо лише не null-значення
val f5 = listOfAny.filterNotNull()
println(f5)
```

Крім фільтрації, розглянемо декілька функції, що перевіряють умову для всіх елементів списку:

```
//отримуємо true, якщо хоча б один елемент задовольняє
↪  умову
val f6 = list.any { it.parts.isNotEmpty() }
println(f6)
//отримуємо true, якщо жоден з елементів не задовольняє
↪  умову
val f7 = list.none { it.parts.isNotEmpty() }
println(f7)
```

```
//отримуємо true, якщо всі елементи задовольняють умову
val f8 = list.all { it.parts.isNotEmpty() }
println(f8)
```

Kotlin дає змогу зручно групувати елементи колекції, створюючи на їх основі асоціативний список (`Map`).

```
val list = listOf("a", "b", "c", "d", "d")
//групуємо елементи, повертаючи в лямбда-функції ключ:
↪  {A=[a], B=[b], C=[c], D=[d, d]}
val res1 = list.groupBy { key -> key.toUpperCase() }
println(res1)
```

Як бачимо із прикладу, як значення у результуючому асоціативному списку ми маємо колекцію елементів, які підпадають під відповідну умову у лямбді.

Якщо потрібно зробити вибірку елементів, у бібліотеці Kotlin є функції `take`, `drop`, `slice`, `chunked` та `windowed`. Розглянемо їх застосування на прикладах:

```
val list = listOf("a", "b", "c", "d", "d")
//отримуємо елементи у заданому інтервалі: результат [a,
↪  b, c]
val sliced = list.slice(0..2)
println(sliced)
//розділяємо колекцію на декілька, згідно розміру:
↪  результат [[a, b], [c, d], [d]]
val chunked = list.chunked(2)
println(chunked)
//розділяємо колекцію методом "вікна" певного розміру:
↪  результат [[a, b], [b, c], [c, d], [d, d]]
val windowed = list.windowed(2)
println(windowed)
//беремо тільки перші три елементи: результат [a, b, c]
val t1 = list.take(3)
println(t1)
```

```
//пропускаємо один елемент спочатку: результат [b, c, d,
↪  d]
val d1 = list.drop(1)
println(d1)
//беремо останні два елементи: результат [d, d]
val t2 = list.takeLast(2)
println(t2)
//пропускаємо три останні елементи: результат [a, b]
val d2 = list.dropLast(3)
println(d2)

//беремо перші елементи згідно умови: результат [a]
val t3 = list.takeWhile { it == "a" }
println(t3)
//пропускаємо перші елементи згідно умови: результат [b,
↪  c, d, d]
val d3 = list.dropWhile { it == "a" }
println(d3)
//аналогічно для takeLastWhile та dropLastWhile
```

1.6.6. Пошук, сортування та об'єднання колекцій

Для пошуку елемента в колекції використовують функцію `find`/ `findLast`.

```
val list = listOf("a", "b", "c", "d", "d")

//повертає перший знайдений елемет згідно умови: d
val f1 = list.find { it == "d" }
println(f1)
//повертає останній знайдений елемет згідно умови: d
val f2 = list.findLast { it == "d" }
println(f2)
```

Kotlin надає зручні інструменти для отримання елементів із колекцій, зокрема знаходження першого/останнього елемента за

зазначеною умовою, а також випадкового елемента.

```
val list = listOf("a", "b", "c", "d", "d")

//поверне перший елемент або NoSuchElementException
val f1 = list.first()
println(f1)
//поверне перший елемент за заданою умовою або
↪  NoSuchElementException
val f2 = list.first{ it == "c" }
println(f2)
//поверне перший елемент за заданою умовою або null
val f3 = list.firstOrNull{ it == "f" }
println(f3)

//поверне останній елемент або NoSuchElementException
val l1 = list.last()
println(l1)
//поверне останній елемент за заданою умовою або
↪  NoSuchElementException
val l2 = list.last{ it == "b" }
println(l2)
//поверне останній елемент за заданою умовою або null
val l3 = list.lastOrNull{ it == "d" }
println(l3)

//отримання випадкового елементу або
↪  NoSuchElementException, якщо список пустий
val r1 = list.random()
println(r1)
//отримання випадкового елементу або null, якщо список
↪  пустий
val r2 = list.randomOrNull()
println(r2)
```

Якщо колекція містить елементи `Comparable` типу, до неї можна застосувати методи натурального сортування, тобто функції

`sorted` та `sortedDescending`. Відсортувавши таким способом колекцію, отримаємо відсортований варіант, `sortedDescending` сортує елементи в порядку спадання.

```
val list = listOf(2, 4, 6, 3, 5)  //[2, 3, 4, 5, 6]
println(list.sorted())
```

Існують також варіанти `sortedBy` та `sortedByDescending`, де можна задати функцію-селектор у разі відсутності у елементів реалізації `Comparable`.

```
val list = listOf("aaaa", "a", "aa",
↪  "aaaa")        //[a, aa, aaaa, aaaa]
println(list.sortedBy { it.length })
```

Крім того, можемо використати функції для легкого створення списку, у якому елементи розміщені у зворотному порядку або перемішані.

```
val list = listOf(1, 2, 3, 4, 8, 5)
//список у зворотньому порядку: [5, 8, 4, 3, 2, 1]
println(list.reversed())
//список у зворотньому порядку, проте всі зміни також
↪  будуть відображені: [5, 8, 4, 3, 2, 1]
println(list.asReversed())
//перемішує список [5, 1, 3, 8, 4, 2]
println(list.shuffled())
```

Для знаходження мінімального або максимального елемента колекції використовують функції `minOrNull` та `maxOrNull`, відповідно, а також `minByOrNull` та `maxByOrNull` з функцією-селектором. Крім того, існує версія `minWithOrNull` та `maxWithOrNull`, у якій можна вказати власний `Comparator` та отримати, відповідно, мінімальний і максимальний елемент. У разі неможливості знайти шукане значення, функція повертає `null`.

```
val list = listOf(1, 2, 3, 4, 8, 5)
//знаходимо мінімальний елемент або null: 1
println(list.minOrNull())
//знаходимо максимальний елемент або null: 8
println(list.maxOrNull())
//мінімальний елемент для функції-селектора, що повертає
↪  найменше значення: 4
println(list.minByOrNull{ it < 4 })
//порівняння використовуючи Comparator: 1
println(list.minWithOrNull( compareBy { it } ))
```

Kotlin дає змогу без лишніх зусиль знайти суму значень, середнє значення серед елементів колекції, підрахувати кількість тощо.

```
val list = listOf(1, 2, 3, 4, 8, 5)
//знаходження середнього значення серед елементів
↪  списку: 3.8333333333333335
println(list.average())
//підрахунок елементів списку: 6
println(list.count())
//сумування елементів списку: 23
println(list.sum())
//сумування з функцією-селектором: 29
println(list.sumOf { it + 1 })
```

Зверніть увагу, що для функції `count` є також варіант із функцією-селектором, що дає можливість більш гнучко виконати підрахунок елементів.

Ще одним цікавим засобом стандартної бібліотеки Kotlin є функції `reduce` та `fold`. Вони дають змогу акумулювати елементи списку в один. Різниця між ними лише в тому, що `fold` використовує для ініціалізації вхідний елемент, водночас як `reduce` використовує для цього перший та другий елемент.

```
//Reduce:
val listOfVehicles = listOf(Vehicle(100.0),
↪  Vehicle(150.0), Vehicle(200.0), Vehicle(280.0))
val reduce = listOfVehicles.reduce{total, v->
        total.speed += v.speed
        total }
//отримаємо об'єкт із значенням швидкості 730.0
println(reduce.speed)

//Fold:
val fold = listOfVehicles.fold(Vehicle(10.0)){total, v->
        total.speed += v.speed
        total }
//отримаємо об'єкт із значенням швидкості 740.0
println(fold.speed)
```

Із цього прикладу бачимо, що у разі використання fold ми отримали об'єкт із значенням швидкості 740.0, оскільки для ініціалізації використовували об'єкт зі значенням 10.0. Існують також різновиди функції, такі як: `Indexed`(з доступом до індексу елемента), `Right`(прохід з кінця колекції) та `runningReduce`/ `runningFold` (для збереження проміжного результату).

1.7. Багатопотоковість

1.7.7. Створення потоків у Kotlin

Потоки у Kotlin можна створити двома способами: за допомогою створення класу, що наслідується від `Thread`, або за допомогою функції `thread`. Розглянемо два випадки:

```
fun main(args: Array<String>) {

        //створюємо екземпляр класу CalculationThread
    val cThread1 = CalculationThread()
    //запускаємо потік на виконання
```

```
    cThread1.start()
        //використовуємо Thread та передаємо Runnable
        val cThread2 = Thread{ println("Running thread
↪   2!") }
        //запускаємо потік на виконання
        cThread2.start()
        //використовуємо функцію thread
        thread(start = true) {
            println("Running thread 3!")
          }
}

class CalculationThread: Thread(){
        override fun run() {
            println("Running thread 1!")
    }
}
```

Завдяки виконанню прикладу отримуємо три незалежні завдання:
Running thread 1!
Running thread 2!
Running thread 3!

Функція `thread` може приймати декілька параметрів, один з яких був використаний у прикладі для негайного запуску:

`start` – негайно запускає потік;

`name` – задає ім'я потоку;

`priority` – задає пріоритет потоку;

`contextClassLoader` – об'єкт `ClassLoader` для використання або завантаження класів та ресурсів;

`isDaemon` – маркує процес як демон.

1.7.7. Корутини (Coroutines)

З першого погляду здається, що ми ні в чому не обмежені та можемо створювати безліч потоків для виконання тих чи інших паралельних завдань. Проте такий підхід призведе до значного сповільнення виконання програми, неможливості створити потрібну кількість потоків тощо. Для більш зручного управління асинхронними завданнями Kotlin пропонує новий підхід – `Coroutines` (корутини). Створення корутин не потребує значних затрат ресурсів, проте вони мають усі потрібні можливості для використання у паралельних завданнях.

Корутини можуть існувати лише в певному контексті `CoroutineScope`. Наприклад, запустивши корутину в `GlobalScope`, її час життя буде обмежено життям застосунку. Створимо та запустимо просту корутину:

```
fun main(args: Array<String>) {

    println("Main start")

        GlobalScope.launch {
        println("Coroutine start!")
        delay(1000L)
        println("Coroutine end!")
        }
        println("Main end")
}
```

Отже, отримаємо:

```
Main start
Main end
```

Як бачимо, корутина не змогла запуститись, оскільки сама програма завершилась раніше. Модифікуємо код, додамо затримку.

```
Thread.sleep(3000L)
println("Main end")
```

Отримуємо:
Main start
Coroutine start!
Coroutine end!
Main end

За допомогою функції `delay` можна призупинити корутину, не блокуючи при цьому основний потік, з якого ця корутина викликається. Використовуючи функцію `runBlocking`, ми можемо створити корутину, проте основний потік, з якого буде викликана ця функція, чекатиме на її виконання.

```
println("Main start")

runBlocking {
        println("Coroutine start!")
        delay(1000L)
        println("Coroutine end!")
        }
println("Main end")
```

Отримуємо очікуваний результат:
Main start
Coroutine start!
Coroutine end!
Main end

Цікавою особливістю функції `delay` всередині корутини є те, що вона позначена як `suspend`. Використання цього ключового слова під час оголошення функції говорить про те, що вона може призупиняти виконання корутини. Таким способом, можна створювати власні `suspend` функції та викликати в них наявні.

Для створення корутини, що повертатиме результат виконан-

ня, можна використати функцію `async`.

```
fun main(args: Array<String>) = runBlocking {
        println("Main start")

        val result = async {
        delay(5000L)
        "async coroutine"
        }
        println(result.await())
        println("Main end")
}
```

Отримаємо:
Main start
async coroutine
Main end

Як бачимо з прикладу, async повертає об'єкт типу `Deferred<T>`, який повертається з корутини як результат. У цьому випадку це `Deferred<String>`. За допомогою функції `await` очікується та повертається результат виконання. Альтернативно код можна записати так:

```
println("Main start")

val result = async {
        delay(5000L)
        "async coroutine"
}

result.join()

println(result.getCompleted())
println("Main end")
```

Корутини повертають об'єкт типу `Job`, що є виконуваною дією, котра може бути відмінена, а також набувати різних станів, завершуючись після виконання всього блоку коду. Як бачимо із прикладу, для того, щоб дочекатись результату, було викликано функцію `join`, тобто `runBlocking`, що її викликає, чекатиме завершення роботи корутини. Завдяки корутинам можна запускати десятки тисяч операцій, не витрачаючи на це велику кількість ресурсів.

Корутини не належать до стандартної бібліотеки, для їх використання потрібно використати бібліотеку Kotlin.

build.gradle.kts

```
dependencies {
        implementation("org.jetbrains.kotlinx:
↪  kotlinx-coroutines- core:1.5.2")
}
```

Контрольні запитання та завдання

1. Чи можна використати ключове слово lateinit для оголошення змінних стандартних типів (Int, Long, Double тощо)?
2. Наведіть приклад класу з декількома конструкторами.
3. Для чого використовують ключове слово operator?
4. З якою ціллю використовують функцію let, apply?
5. Як можна зручно створити клас-синглтон?
6. Чи можна створити клас всередині функції?
7. Що дає можливість отримати використання ключового слова reified?
8. За допомогою якого ключового слова позначають перевизначення функції/властивості у класі?
9. Що таке suspend функція? Наведіть приклад використання.

10. Яка відмінність між List та MutableList?
11. Як можна достроково завершити процес ітерації forEach?
12. Наведіть приклад використання функції-розширення.
13. Для чого застосовують ключове слово where? Наведіть приклад.
14. Напишіть приклад функції з використанням параметризованих типів.
15. Як працює функція thread? Наведіть приклад.

Розділ 2.

Kotlin-ресурси для роботи з даними

2.1. Інтерактивні редактори для роботи з даними

Крім інтегрованого середовища розробки, наприклад, Intellij Idea (див. розділ 1.1.) яке може виконувати всі завдання розробки в одному інструменті, варто згадати про подібні інструменти, які називають записниками (notebook). Блокноти (записники) дають можливість користувачам проводити дослідження даних, їх обробку та зберігати результати в єдиному середовищі. У блокноті Ви можете написати описовий текст поруч із блоком коду, виконати блок коду та переглянути результати в будь-якому потрібному Вам форматі: вихідний текст, таблиці, візуалізація даних тощо. Kotlin забезпечує дуже легку інтеграцію з двома популярними блокнотами: Jupyter Notebook і Apache Zeppelin, які дають змогу писати та запускати блоки коду Kotlin, що значно полегшує процес аналізу даних.

Одним із інструментів, який може допомогти вам ітеративно досліджувати, візуалізувати та ознайомлюватися з Вашими даними, є Jupyter Notebook [1]. Jupyter Notebook це веб-

програма з відкритим вихідним кодом, яка допомагає створювати та ділитися документами (блокнотами), які можуть містити код, візуалізації та текст розмітки. Блокноти складаються з комірок, які можна запускати по черзі, різні типи комірок можна комбінувати в одному блокноті. Наприклад, Ви можете доповнити свій код (дослідження) розповіддю та рівняннями.

Jupyter Notebook є надзвичайно популярним інструментом у спільноті Data Science. Він досить універсальний і може слугувати багатьом цілям: його можна використовувати для навчання, обробки й аналізу даних, створення звітів для бізнес-стейкхолдерів, навчання моделям машинного навчання тощо. Є декілька причин популярності Jupyter:

- Інтерактивне середовище зручне для швидких експериментів: незалежно від того, чи Ви пробуєте бібліотеку, чи знайомитеся з новим набором даних, чи випробовуєте кілька алгоритмів машинного навчання, Ви можете швидко повторювати свої ідеї в блокноті.

- Його легко налаштувати, і Ви можете запустити Jupyter Notebook локально або на потужному віддаленому комп'ютері для виконання ресурсомістких обчислень.

- Jupyter спрощує візуалізацію даних: дослідження розподілу ваших даних, візуалізації тенденцій, дослідження кореляції тощо.

- Блокноти допомагають легко ділитися своїми висновками з колегами. Нарешті, Jupyter підтримує кілька мов програмування: Python, R, Scala, Java, а тепер і Kotlin.

Хоча Jupyter Notebook чудово підходить для великої кількості завдань, жоден інструмент не є ідеальним. Важливо розуміти його обмеження, а також його силу. Ось кілька речей, про які варто пам'ятати під час використання Jupyter Notebook:

- Jupyter дає можливість виконувати комірки коду не за порядком, що швидко створює безлад.

- Найпоширеніші методи розробки програмного забезпечення, як-от тестування, налагодження та контроль версій, є досить складними, коли йдеться про Jupyter Notebook, що робить його непридатним для робочого коду.

- Якщо Ви звикли до розширених функцій IDE, таких як навігація, аналіз помилок, завершення коду, пошук документації тощо, більшість ядер Jupyter матимуть лише невелику частину таких функцій, що може засмучувати.

Іншими словами, як і будь-який інший інструмент, за правильного використання, Jupyter Notebook може стати чудовим доповненням до Вашого арсеналу інструментів.

Якщо Ви хочете досліджувати набір даних, розвивати навички обробки даних і машинного навчання або грати з новими бібліотеками, але не хочете залишати комфорт світу JVM у обмін на динамічний Python, Ви можете робити все це за допомогою Kotlin Kernel для Jupyter Notebook.

Щоб налаштувати Jupyter Notebook, спочатку Вам потрібно встановити Python на вашому комп'ютері. Інструменти Project Jupyter доступні для встановлення через Python Package Index, провідне сховище програмного забезпечення, створеного для мови програмування Python. Існує дві реалізації Jupyter Notebook – класична та більш досконала jupyter-lab. Основна відмінність між ними це інтерфейс користувача. Більш детально з установленням Jupyter Notebook можна ознайомитись тут [1].

Щоб спробувати ядро Kotlin із наявним блокнотом Jupyter, установіть його за допомогою conda або pip install:

conda: conda install kotlin-jupyter-kernel -c jetbrains

pip install: pip install kotlin-jupyter-kernel

Kotlin Jupyter потребує встановлення Java 8.

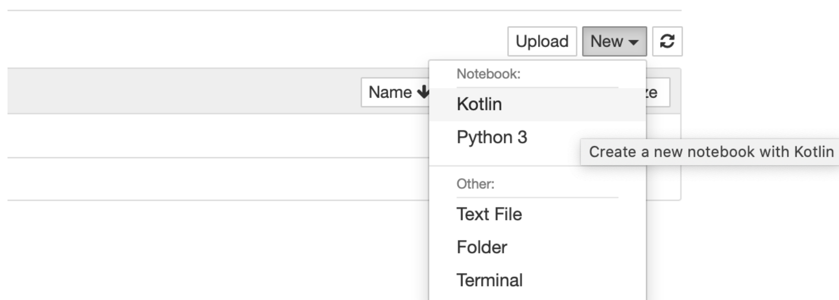
Почніть свій блокнот Jupyter з

```
jupyter notebook
```

або

```
jupyter-lab
```

Щойно Jupyter Notebook запуститься та відкриється у Вашому браузері, Ви готові почати створювати нові блокноти з ядром Kotlin.



Ядро Kotlin підтримує чимало бібліотек, які зазвичай використовують для роботи з даними, Ви можете почати використовувати ці бібліотеки, додавши просту «магію лінії».

```
%use [library]
```

Після цього Ви можете додавати комірки з Kotlin-кодом, виконувати його та спостерігати результати роботи. У наступних розділах наведено приклади використання Jupyter Notebook.

Також варто згадати про ще одну реалізацію блокнотів Jupyter. Datalore – це потужне онлайн-середовище для записників Jupyter, яке допомагає більш продуктивно редагувати, виконувати та ділитися своїм кодом [2]. Без необхідності налаштування Ви можете використовувати Datalore для збору та дослідження даних, машинного навчання, глибинного навчання та інтерактивної візуалізації.

Datalore реалізує такі функції:

- Готові до використання інструменти науки про дані: науково-популярні бібліотеки даних уже попередньо встановлено, щоб Ви могли швидко розпочати роботу з ноутбуками Jupyter.

- Редактор дозволяє писати код безпосередньо в браузері з повною підтримкою Python, Kotlin, R і Scala. Редактор надає допомогу в кодуванні, доповненні коду, параметрах перевірки та рефакторингу, автоматичних швидких виправлення, підказках та інших параметрах.

- Ви можете ділитись своєю роботою зі своєю командою, відстежувати прогрес за допомогою вбудованої системи контролю версій і надавати коментарі в текстових комірках із підтримкою Markdown і LaTex.

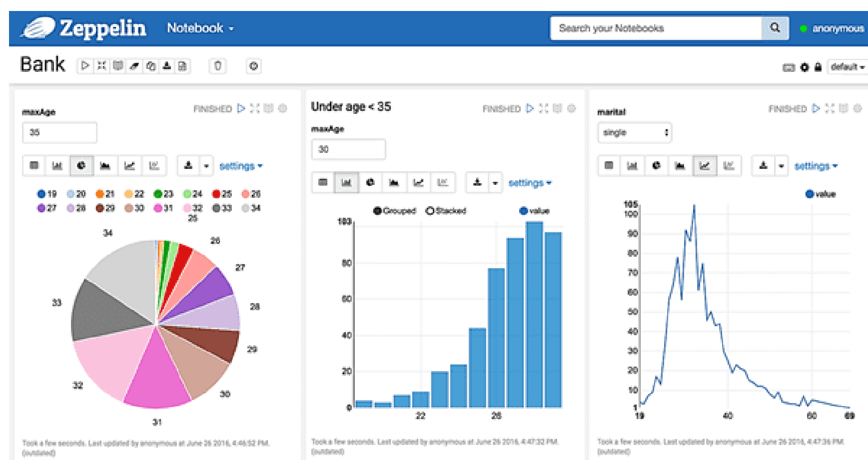
Приклади коду в цих процедурах написані мовою Python, мовою програми за замовчуванням. Однак Datalore також забезпечує підтримку Kotlin, доступний через вебінтерфейс [2].

Досить близький аналог Jupyter Notebook – це Apache Zeppelin [3]. Своєю чергою Zeppelin дещо більш розрахований для роботи з базами даних. Він використовує концепцію «інтерпретаторів» – плагінів, які забезпечують бекенд для будь-якої мови та/або бази даних. Zeppelin, як і Jupyter, для користувача є як набором файлів-ноутбуків, що складається з параграфів, у яких пишуть та виконують запити. За допомогою вбудованих візуалізаторів ноутбук із набором запитів легко перетворити на повноцінний дашборд з даними. Процес встановлення досить докладно описаний на сторінці виробника [3].

Отже, Apache Zeppelin – це інтерактивний вебблокнот з відкритим вихідним кодом, який підтримує практично всі етапи роботи з даними – від вилучення до візуалізації – в тім числі інтерактивний аналіз та спільне використання документів. Він інтегрований з Apache Spark, Flink, Hadoop, безліччю реляційних і NoSQL-СУБД (Cassandra, HBase, Hive, PostgreSQL, Elasticsearch,

Google Big Query, MySQL, MariaDB, Redshift), а також підтримує різні мови програмування, популярні в області Big Data: Python, PySpark, R, Scala, Kotlin, SQL. Така багатофункціональність забезпечується завдяки плагінам інтерпретаторів для підтримки мови програмування, бази даних чи фреймворку.

З точки зору роботи з великими даними, на окрему згадку заслуговує вбудована інтеграція з Apache Spark, що дає загальні контексти (SparkContext і SQLContext), завантаження jar-залежностей з локальної файлової системи або репозиторія maven під час виконання завдання, а також можливість скасування завдання та відображення ходу його виконання. Також Zeppelin підтримує роботу з REST-API Apache Spark – Livy. Завдяки інтерпретатору Kotlin Apache Zeppelin надає всі можливості цієї мови, орієнтовані на аналітику великих даних та візуалізацію. Це дає можливість автоматично побудувати кругові, стовпчасті та інші наочні діаграми, щоб візуалізувати статистику даних чи результатів дослідження. Також у Zeppelin можна створювати інтерактивні дашборди з формами введення даних, які виглядатимуть як веб-сторінки, щоб поділитись їх URL-адресами для спільної роботи. Для розрахованого на багато користувачів режиму Zeppelin підтримує LDAP-авторизацію з налаштуваннями доступу.



,

Утім, за всіх цих переваг, на практиці можна зіткнутися з такими обмеженнями Apache Zeppelin, які можна розглядати як недоліки:

- нестабільна робота під високим навантаженням;
- інтерактивний вебінтерфейс потребує багато оперативної пам'яті;
- відсутність повного набору можливостей сучасних спеціалізованих IDE;
- менша «зрілість» та популярність порівняно з Jupyter Notebook.

Проте, Apache Zeppelin завойовує свою нішу, конкуруючи з Jupyter Notebook у деяких випадках роботи з великими даними.

Apache Zeppelin vs Jupyter Notebook. Насамперед зазначимо, що обидва інструменти належать до open-source і є вебблокнотами для розробки та візуалізації даних. Однак Jupyter позиціонується як багатомовне інтерактивне обчислювальне середовище, з підтримкою коду, рівнянь, текстів, графіків та інтерактивних дешбордів. Apache Zeppelin не претендує на лаври IDE, хоча включає деякі функції для розробки ПЗ, фокусуючись на можливостях для інтерактивного аналізу великих даних. Розберемо, як обидва блокноти відрізняються за такими критеріями, важливими з точки зору роботи з Big Data:

безпека та розраховані на багато користувачів можливості
: які Jupyter не підтримує за замовчуванням, на відміну від Zeppelin. Крім того, у Jupyter немає можливості забезпечення конфіденційності кінцевих користувачів. Zeppelin допомагає гнучко налаштовувати конфігурації безпеки, включаючи конфіденційність програмного коду через LDAP/Active Directory та спеціально визначені групи безпеки. Він використовує лише один серверний процес, аутентифікуючи користувачів у налаштованій системі, перш ніж дозволити подаль-

ший доступ, щоб ділитися інформацією лише з обмеженим колом осіб із певними правами.

візуалізація завдяки можливості використовувати різні інтерпретатори в одному блокноті Zeppelin виграє порівняно з Jupyter, у якому немає параметрів побудови діаграм. У Jupyter є бібліотеки, які виводять діаграму в блокнот, тоді як Zeppelin підтримує лише вміст Matplotlib – Python-бібліотеку побудови двовимірних графіків, яка просто зберігає виведення у HTML-файлі.

опис звітів обидва інструменти підтримують markdown-розмітку, але Zeppelin швидше створює інтерактивні форми та візуалізацію результатів. Крім того, цепелін-звіти більш доступні для кінцевих користувачів і можуть бути експортовані у формат CSV або TSV. Zeppelin допомагає приховати код, надаючи інтерактивні звіти, що читаються, кінцевим користувачам.

кластерна інтеграція Zeppelin є частиною екосистеми Apache Hadoop та добре інтегрується зі Spark, Pig, Hive та іншими її компонентами.

зручність розробки на відміну від Jupyter, Zeppelin дає змогу комбінувати кілька параграфів в один рядок, проте редактор коду та комірок у Jupyter є більш ефективними, оскільки мають більше швидких комбінацій (гарячих клавіш) та функцію автозаповнення.

виробнича експлуатація (production) оскільки Zeppelin залежить від ємності кластера, то за нестачі ресурсів або великій кількості користувачів (понад 100), можливі збої та зависання, які не характерні для Jupyter.

Підсумовуючи, зазначимо, що Apache Zeppelin – чудовий інструмент для аналітики великих даних у екосистемі Hadoop. Він спрощує розробку Spark-додатків та орієнтований на корпоративних користувачів, забезпечуючи інтеграцію з LDAP, управління дозволами та інтерактивну візуалізацію за достатньої кількості

ресурсів кластера. Своєю чергою Jupyter Notebook потребує менше накладних витрат на налаштування та створення розроблених шаблонів завдяки автономному характеру. А завдяки великій кількості IDE-функцій, розширень та підтримці фреймворків машинного навчання та інших методів штучного інтелекту він став дуже популярним серед індивідуальних Data Science дослідників.

2.2. Бібліотеки

Екосистема бібліотек для завдань, пов'язаних з даними, створена спільнотою Kotlin, стрімко розширюється. Розроблено чимало бібліотек для статистичних досліджень, візуалізації даних, роботи зі штучним інтелектом. Більшість зі згадуваних нижче бібліотек мультиплатформні, відповідно, вони можуть бути використані в додатках як із підтримкою JVM, так і в нативних програмах. Ось деякі бібліотеки, які можуть бути Вам корисними.

2.2.2. Lets-Plot: графіка для статистичних даних

Ви можете багато зрозуміти про дані з показників, перевірок та базової статистики. Проте, як люди, ми набагато швидше сприймаємо тенденції та закономірності, коли бачимо їх на власні очі. Тому насамперед, на нашу думку, потрібно згадати бібліотеки які забезпечують візуалізацію даних. Однією з таких бібліотек є lets-plot [4], це єдина мультиплатформна бібліотека. Завдяки унікальній мультиплатформнній природі Kotlin, функція побудови графіків написана один раз у Kotlin, а потім може бути запакована, як бібліотека JavaScript, бібліотека JVM і, власне, розширення Python.

Lets-Plot для Kotlin – це бібліотека з відкритим вихідним кодом для статистичних даних, повністю написана на Kotlin. Розглянемо її архітектуру, види графіків, які можна створювати за її допомогою, і те, що робить цю бібліотеку унікальною.

Бібліотека поширюється через репозиторій Maven. Ви можете включити його у свій проєкт Kotlin за допомогою файлів конфігурації Maven або Gradle або включити його до свого сценарію записника jupyter за допомогою анотації `%use lets-plot` [5].

Якщо рахувати попередні зауваження (розділ 2.1.) радимо розпочати знайомство з бібліотекою lets-plot використовуючи записник Jupiter Notebook. Це дасть змогу пришвидшити процес, позбувшись нескінчених компіляцій коду.

lets-plot API подібний до ggplot і створений з урахуванням принципів багатошарової графіки. Ви можете бути знайомі з цим підходом, якщо коли-небудь використовували пакет ggplot2 для мови R.

> *"Ця граматика ... складається з набору незалежних компонентів, які можна складати різними способами. Це стає дуже потужним інструментом, оскільки Ви не обмежені набором попередньо визначеної графіки, але Ви можете створювати нову графіку, яка точно підходить для Вашої проблеми". Hadley Wickham* [6].

Розуміння архітектури Lets-Plot. У Lets-Plot графік зображено принаймні одним шаром. Шари відповідають за створення об'єктів, намальованих на «полотні», і містять такі елементи:

Data – підмножина даних, визначена один раз для всіх шарів або для кожного рівня. Один графік може поєднувати кілька різних наборів даних (по одному на шар).

Aesthetic mapping – описує, як змінні в наборі даних зіставляються з візуальними властивостями шару, такими як колір, форма, розмір або положення.

Geometric object – геометричний об'єкт, що є певним типом діаграми.

Statistical transformation – обчислює будь-який статистичний підсумок на необроблених вхідних даних. Наприклад, статистику *bin* використовують для гістограм, а гладку – для ліній регресії.

Position adjustment – метод, який застосовують для обчислення кінцевих координат геометрії. Використовують для побудови варіантів одного геометричного об'єкта або для уникнення накладання об'єктів.

Щоб об'єднати усі ці частини разом, потрібно використовувати таку просту формулу:

```
p = letsPlot(<dataframe>)
p + geom<chart_type>(stat=<stat>, position=<adjustment>)
        {<aesthetics mapping> }
```

Розглянемо простий приклад, типу "Hello world!"з бібліотекою lets-plot, отже, в записнику jupiter:

In [1]:

```
//Змусить ядро Kotlin отримати та застосувати
↪  останню версію сховища
%useLatestDescriptors
//Включити весь необхідний шаблонний код
↪  Lets-Plot до блокнота
%use lets-plot
```

Наступним кроком потрібно заповнити структуру даних для відображення. Дані в lets-plot потрібно подати у вигляді асоціативного масиву ключ-значення, загалом

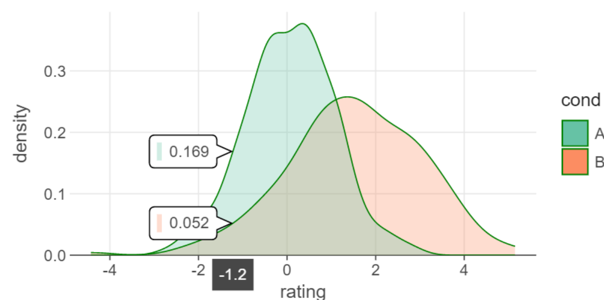
```
mapOf<String, Any>("x" to <List>, "y" to <List>)
```

Отже,

In [2]:
```
val rand = java.util.Random(123)
val n = 200
val data = mapOf<String, Any>(
    "cond" to List(n) { "A" } + List(n) { "B"
↪  },
    "rating" to List(n) { rand.nextGaussian()
↪  } + List(n) { rand.nextGaussian() * 1.5 +
↪  1.5 },
)
//Відображення згенерованих даних
var p = letsPlot(data)
p += geomDensity(color="dark_green",
↪  alpha=.3) {x="rating"; fill="cond"}
p + ggsize(500, 250)
)
```

За відображення даних відповідає функція `letsPlot(): Plot`, яка фактично виконує всю роботу з інтерпретації попередньо сформованих даних.

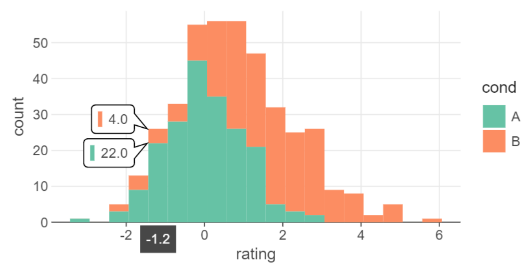
Out [2]:



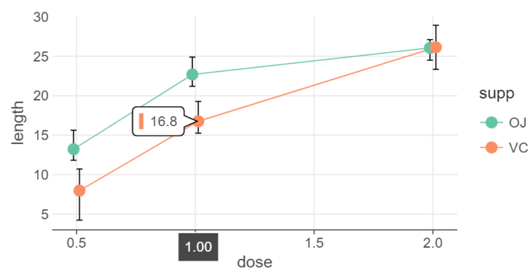
У наведеному прикладі використано тип відображення `geomDensity()`, однак бібліотека надає значний перелік типів діаграм.

Різновиди діаграм (Geometric objects). Існує певна кількість базових видів діаграм в lets-plot. Ви можете додати новий геометричний об'єкт (або шар графіки), створивши його за допомогою функції `geomXxx()`, а потім додати цей об'єкт до діаграми. Різновиди діаграм доступні в lets-plot:

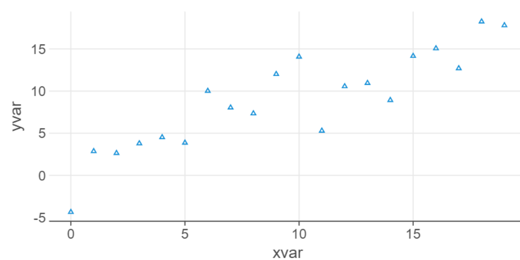
- **Histograms and bar plots** – гістограми та стовпчасті діаграми
  \+ `geomHistogram()`.



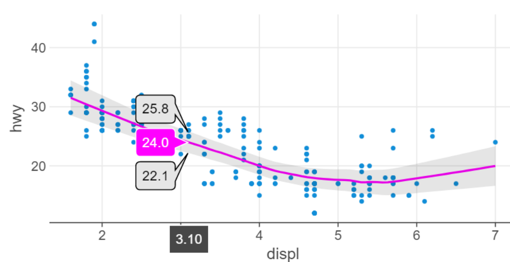
- **Error bars** – смуги похибок
  \+ `geomErrorBar()`.



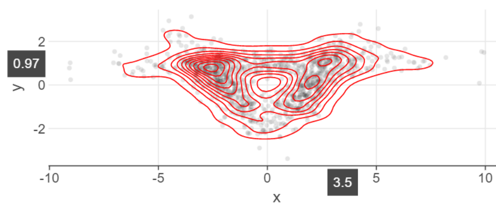
- **Scatter plots** – діаграми розсіювання
  `+ geomPoint()`.



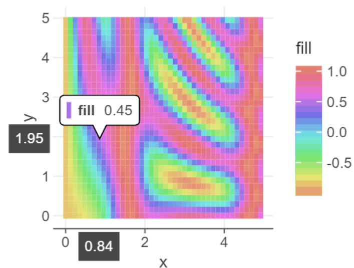
- **Smoothing** – згладжування
  `+ geomPoint() + statSmooth()`.



- **Density plots** – графіки щільності
  `+ geomDensity2D()`.



- **Contour plots** – контурні ділянки
  + `geomTile()` + `scaleFillHue()`.



- **Maps** – карти
  + `geomPolygon()`.

Налаштування діаграм. З коробки lets-plot підтримує вищеперелічені типи візуалізації. Усі графіки гнучкі та легко налаштовуються, проте бібліотеці вдається зберегти баланс між потужними можливостями налаштування та простотою використання [7].

Статистика. Для використання статистичної інформації у ваших діаграмах додайте `stat` як аргумент до функції `geomXxx()`, щоб визначити перетворення статистичних даних:

```
geomPoint(stat=Stat.count())
```

Підтримувані статистичні перетворення:

identity: залишити дані без змін;

count: обчислити кількість точок з однаковою координатою осі x;

bin: обчислити кількість точок, що потрапляють у кожен із суміжних діапазонів однакового розміру вздовж осі x;

bin2d: обчислити кількість точок, що потрапляють у кожен із суміжних прямокутників однакового розміру на площині графіка;

smooth: виконати згладжування;

contour, contourFilled: обчислити контури 3D-даних;

boxplot: обчислити компоненти стовпчастої діаграми;

density, density2D, density2DFilled: виконати оцінку щільності ядра для 1D і 2D даних.

Естетика відображення. За допомогою зіставлення можна визначити, які змінні в наборі даних відповідають візуальним елементами діаграми. Додайте `{x=< >; y=< >; ...}` до геометричного об'єкта, де:

x: стовпець кадру даних для відображення на вісь x;

y: стовпець фрейму даних для зіставлення з віссю y.

...: інші візуальні властивості діаграми, такі як колір, форма, розмір або положення.

Наприклад:

```
geomPoint() {x = "cty"; y = "hwy"; color="cyl"}
```

Регулювання положення. Усі шари мають коригування положення, яке обчислює кінцеві координати геометрії. Зміну позиції використовують для побудови однакових типів графіків і вирішення проблем перекриття. Ви можете перевизначити параметри за замовчуванням за допомогою зміни аргументу position у функціях `geomXxx`:

```
geomBar(position=positionFill)
```

або

```
geomBar(position=positionDodge(width=1.01))
```

Доступні налаштування: `dodge`, `jitter`, `jitterdodge`, `nudge`, `identity`, `fill`, `stack`.

Особливості, що впливають на весь графік

Шкали (Scales) – ця опція допомагає вибрати розумний масштаб для кожної відображеної змінної залежно від її атрибутів.

Ви можете перевизначити шкали за замовчуванням, щоб налаштувати такі деталі, як мітки осей або ключі легенд, або використати зовсім інший підхід для естетичного відображення даних. Наприклад, щоб змінити колір заливки на гістограмі:

```
p + geomHistogram() + scaleFillContinuous("red",
↪  "green")
```

Система координат визначає, як естетика x і y поєднується для позиціонування елементів на діаграмі. Наприклад, щоб змінити стандартне співвідношення X і Y:

```
p + coordFixed(ratio=2)
```

Легенда[1] *(Legend).* Осі та легенди допомагають користувачам інтерпретувати графіки. Використовуйте методи `guide` або аргумент `guide` методу `scale`, щоб налаштувати легенду. Наприклад, щоб визначити кількість стовпців у легенді:

```
p + scaleColorDiscrete(guide=guideLegend(ncol=2))
```

Дивіться більше інформації про функції `guideColorbar`, `guideLegend` у офіційній довідці [8].

Вибірка (Sampling) це спеціальна техніка трансформації даних, вбудована у lets-plot, яку застосовують після обчислення статистики. Вибірка допомагає запобігти зависанню інтерфейсу користувача та збоям у зв'язку з нестачею пам'яті під час спроби побудувати надто велику кількість об'єктів. За замовчуванням метод застосовують автоматично, коли обсяг даних перевищує певний поріг. Значення `samplingNone` вимикає будь-яку вибірку для поточного шару. Методи вибірки можна об'єднати разом за допомогою оператора +.

Доступні методи:

samplingRandomStratified – випадково вибирає точки з кожної групи пропорційно її розміру, але також гарантує, що кожну групу представлено принаймні визначеною мінімальною кількістю точок;

samplingRandom – вибирає точки даних за випадково вибраними індексами без заміни;

samplingPick – аналізує X-значення та вибирає всі точки, для яких X-значення отримують з набору перших n X-значень, знайдених у сукупності;

[1] Легенда (Legend) – це область, де відображено умовні позначення різних даних на діаграмі. Легенда може бути розміщена в різних частинах діаграми: вона може знаходиться як в області діаграми, так і поза її межами.

samplingSystematic – вибирає точки даних за рівномірно розподіленими індексами;

samplingCertexDP, samplingVertexVW – спрощує побудову багатокутників. На вибір є два алгоритми реалізації: Douglas-Peucker (DP) та Visvalingam-Whyatt (VW).

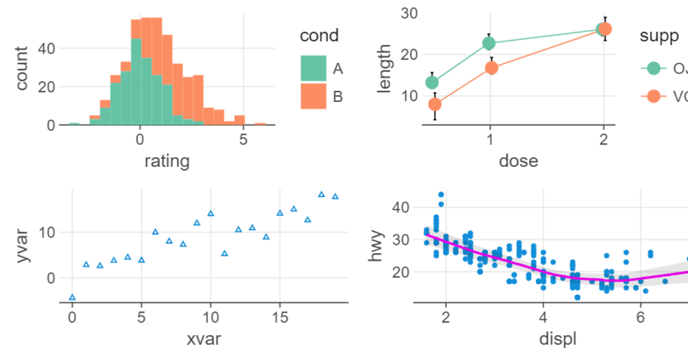
Групування діаграм. За допомогою об'єкта `GGBunch()` Ви можете візуалізувати колекцію графіків. Використовуйте метод `addPlot()`, щоб додати графік до групи та визначити довільне розташування та розмір для графіків усередині сітки:

```
val bunch = GGBunch()
        .addPlot(plot1, 0, 0)
        .addPlot(plot2, 0, 200)
bunch.show()
```

Метод `addPlot()` має ще два (необов'язкові) параметри: ширину та висоту. Ці значення замінять розмір графіка, визначений раніше за допомогою функції `ggsize()`.

In [3]:
```
val lenght = 400
val height = 200
val bunch = GGBunch()
    .addPlot(plot1, 0, 0, lenght, height)
    .addPlot(plot2, lenght, 0, lenght,
↪  height)
    .addPlot(plot3, 0, height, lenght,
↪  height)
    .addPlot(plot4, lenght, height, lenght,
↪  height)
bunch.show())
```

Out [3]:



Lets-Plot в JVM та Kotlin/JS додатках. Основний функціонал бібліотеки lets-plot залишається незмінним як у випадку використання записника jupiter, так і за використання її у додатках. Lets-Plot дає можливість вбудовувати графіки в програму JVM або Kotlin/JS. У середовищі JVM бібліотека lets-plot пропонує вибір між графікою JavaFX і рендерінгом на основі інструментарію Apache Batik SVG [8]. Відповідно, конфігурація проєкту Gradle (Groovy) буде різною для цих платформ. Усі артефакти доступні в репозитарії `mavenCentral()`.

build.gradle.kts

```
repositories {
  mavenCentral()
}
```

Залежності проєкту:

- JVM/Swing/Batik application:

build.gradle.kts

```
dependencies {
  implementation("org.jetbrains.lets-plot:
↪  lets-plot-batik:2.4.0")
  implementation("org.jetbrains.lets-plot:
↪  lets-plot-kotlin-jvm:3.3.0")
}
```

- JVM/Swing/JavaFX application:

build.gradle.kts
```
dependencies {
  implementation("org.jetbrains.lets-plot:
↪  lets-plot-jfx:2.4.0")
  implementation("org.jetbrains.lets-plot:
↪  lets-plot-kotlin-jvm:3.3.0")
}
```

- Kotlin/JS додаток:

build.gradle.kts
```
dependencies {
  implementation("org.jetbrains.lets-plot:
↪  lets-plot-kotlin-js:3.3.0")
}
```

- JVM/інші: Якщо ваша програма JVM не використовує жодного інтерфейсу, ви можете надати лише залежність «lets-plot-common»:

build.gradle.kts
```
dependencies {
  implementation("org.jetbrains.lets-plot:
↪  lets-plot-common:2.4.0")
  implementation("org.jetbrains.lets-plot:
↪  lets-plot-kotlin-jvm:3.3.0")
}
```

у такому випадку найбільш використовуваний варіант використання методу `ggsave()`, який експортує графік у файл. Підтримувані формати: SVG, HTML, PNG, JPEG і TIFF (у деяких конфігураціях растрові формати можуть не підтримуватися). Приклад реалізації такого підходу представлений в додатку A.1., графік

зображений на рис. 2.3. В інших випадках за відображення відповідає метод `show()` в інтерфейсі `Figure`, класи `Plot` і `GGBunch` в Lets-Plot реалізують цей інтерфейс.

Загалом lets-plot має достатньо потужний інструментарій для створення діаграм будь-якої складності, можливо, за винятком тримірних графіків. Ви можете дізнатися більше про основи lets-plot і краще зрозуміти, що роблять окремі будівельні блоки, переглянувши посібник із початку роботи та офіційну документацію [5, 8].

2.2.2. Kravis: бібліотека для візуалізації табличних даних

Продовжуючи тему візуалізації даних, потрібно звернути увагу на ще одну бібліотеку, створену для побудови діаграм – «kravis – A {k}otlin {gra}mmar for data {vis}ualization». Kravis реалізує граматику для створення широкого діапазону візуалізацій, використовуючи стандартизований набір директив [9].

Граматика, реалізована в kravis, натхненна ggplot2. Насправді усе, що він надає kravis – це більш безпечна обгортка навколо ggplot2, всередині якого використовують як механізм візуалізації. API kravis дуже схожий на ggplot2, тому дозволяє навіть повторно використовувати їх чудову шпаргалку.

Мова R є необхідною для використання ggplot2, однак kravis працює з різними інтеграційними серверами, такими як докер або віддалені вебсервіси. На цей момент kravis надає три різні варіанти прив'язки R, який потрібен для візуалізації даних:

Local R – це стандартний режим, який можна налаштувати за допомогою:

```
SessionPrefs.RENDER_BACKEND = LocalR()
```

Dockerized R – за замовчуванням використовує контейнер rocker-tidyverse:3.5.1, однак за потреби його можна налаштувати на використання додаткових образів.

```
SessionPrefs.RENDER_BACKEND = Docker()
```

Rserve – (Необов'язково) віддалений бекенд на основі Rserve,[2] просто встановіть відповідний пакет R і запустіть демон [10]. Крім того, якщо Ви не маєте або не бажаєте локальної інсталяції R, Ви також можете запустити її докеризовано локально або віддалено за допомогою

```
docker run -p <public_port>:<private_port> -d
↪  <image>
```

Щоб використовувати серверну частину Rserve, відповідно, налаштуйте kravis SessionPrefs, указавши правильний хост і порт.

```
SessionPrefs.RENDER_BACKEND =
↪  RserveEngine(host="localhost", port=6302)
```

Детальніше можна ознайомитись у [9].

Простим способом розпочати роботу з kravis є jupyter [1], Вам просто потрібно встановити ядро kotlin-jupyter, приклад блокнота [11]. Для включення kravis до вашого проєкту додайте наступний артефакт до Вашої конфігурації.

build.gradle.kts

```
dependencies {
  implementation("com.github.holgerbrandl:kravis:0.8.5")
}
```

[2]Rserve — це сервер TCP/IP, який дає можливість іншим програмам використовувати засоби R (див. www.r-project.org) з різних мов без необхідності ініціалізації R або зв'язування з бібліотекою R. Кожне підключення має окрему робочу область і робочий каталог. Реалізації на стороні клієнта доступні для таких популярних мов, як C/C++, PHP, JavaScript і Java. Rserve підтримує віддалене підключення, автентифікацію та передання файлів. Типовим використанням є інтеграція серверної частини R для обчислення статистичних моделей, графіків тощо в інших програмах.

Граматика графіки. ggplot2 і, отже, kravis реалізують граматику для графіки, за допомогою якої можна будувати свої діаграми:

```
естетика + шари + система координат +
  трансформації + грані
```

Які читаються як: відобразити змінні з простору даних у візуальний простір + додати один або кілька шарів + налаштувати систему координат + за бажанням застосувати статистичні перетворення + за бажанням додати грані. Так виглядає основна ідея.

Підтримувані формати введення даних

Ітератори Кожен `Iterable<T>` є дійсним джерелом даних для kravis, що дозволяє створювати графіки за допомогою DSL-конструктора[3]. По суті, ми спочатку перетворюємо його в таблицю та використовуємо як джерело даних для візуалізації.

Таблиці Kravis може обробляти будь-які табличні дані через кадри даних (DataFrame) krangl.

Пристрої виведення. Kravis автоматично визначає середовище та намагатиметься вгадати найбільш прийнятний вихідний пристрій для відображення Ваших даних. Доступні такі пристрої виведення:

- swing-графіка для візуалізації під час роботи в інтерактивному режимі;
- графіка javaFX для візуалізації під час роботи в інтерактивному режимі;

[3] Типозахищені конструктори, що допомагають створювати предметно-орієнтовані мови (DSL), придатні для побудови складних ієрархічних структур даних напівдекларативним способом [12, /docs/type-safe-builders.html].

- може візуалізувати безпосередньо у файли та відображатиметься в блокнотах jupyter.

За замовчуванням kravis відображатиметься як png на всіх пристроях, але він також підтримує векторний рендерінг із використанням svg як вихідного формату. Бажаний вихід можна налаштувати за допомогою об'єкта `SessionPrefs`.

```
SessionPrefs.OUTPUT_DEVICE = SwingPlottingDevice()
```

Графіки в бібліотеці kravis є незмінними, подібно до кадрів даних krangl (розділ 2.2.2.).

```
val basePlot = mpgData.plot("displ" to x, "hwy" to
↪  y).geomPoint()

// створити одну версію зі зміненим розміром тексту по
↪  осі
basePlot.theme(axisText = ElementText(size = 20.0, color
↪  = RColor.red))

// створити іншу версію з незмінними мітками осей, але
↪  замість цього використовувати логарифмічний масштаб
basePlot.scaleXLog10()
```

Оскільки kravis просто імітує деякі частини ggplot2 радимо Вам звернутись до документації з API ggplot2, або з проєкту kravis [9]. Користувач може захотіти створити більше власних графіків, kravis підтримує преамбули (наприклад, для визначення нових geom) і специфікації власних шарів.

Приклад програмного коду, в якому спільно використані бібліотека krangl (розділ 2.2.2.) та kravis для візуалізації, результати наведено на рис.2.1, та 2.4. У прикладі використано відомий набір даних sleepData, який містить тривалість сну та ваги для набору видів ссавців. Набір даних містить 83 рядки та 11 змінних. Набір даних поставляється з krangl як приклад (містить 83 рядки та 11 змінних), тому немає необхідності завантажувати його з іншого місця.

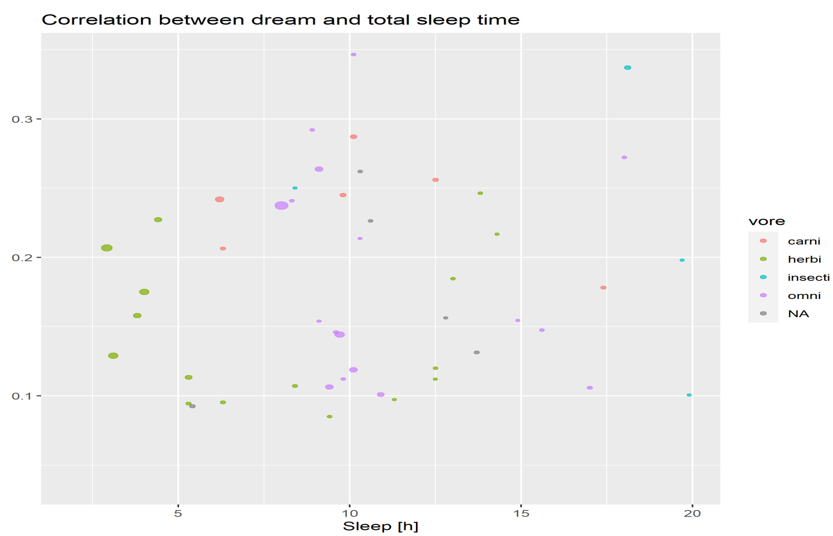


Рис. 2.1. Результат спільного застосування бібліотек kravis та krangl, код програми в додатку A.2.

2.2.2. Multik: бібліотека багатовимірних масивів для Kotlin

Багато завдань, пов'язаних із великим об'ємом даних, а також проблеми оптимізації зводяться до виконання обчислень над багатовимірними масивами, бібліотека Multik, повинна стати основою для таких обчислень. Multik: багатовимірні масиви в Kotlin. Бібліотека надає ідіоматичний API Kotlin, безпечний для типів і розмірів для математичних операцій над багатовимірними масивами. Multik пропонує як JVM так і власні обчислювальні механізми, а також їх комбінацію для оптимальної продуктивності [13].

Бібліотека multik містить оригінальні реалізації структур даних та операцій над ними і не є простою обгорткою для існуючих бібліотек на зразок NumPy [14]. У Multik структури даних відокремлені від виконання операцій над ними, тобто потрібно додати їх як окремі залежності до вашого проєкту. Цей підхід дає Вам послідовний API незалежно від того, яку реалізацію Ви вирішите використовувати у своєму проєкті. Отже, що це за реалізації?

На цей момент існує три різні:

multik-jvm: реалізація математичних операцій у Kotlin/JVM;

multik-native: реалізація C++. OpenBLAS використовують для лінійної алгебри;

multik-default: реалізація за замовчуванням, яка поєднує власну реалізацію та реалізацію JVM для оптимальної продуктивності.

Для використання бібліотеки потрібно включити одну з реалізацій у файл build.gradle.kts, також необхідним є модуль multik-api, який містить об'єкт `ndarrays`, його методи та інтерфейси [math], [stat] і [linalg].

build.gradle.kts

```
repositories {
  mavenCentral()
}
dependencies {
```

```
  implementation("org.jetbrains.kotlinx:multik-api:0.1.1")
  implementation("org.jetbrains.kotlinx:
↪  multik-default:0.1.1")
}
```

Для використання бібліотеки в записнику jupiter потрібно завантажити заголовок.

In [1]:
```
//Змусить ядро Kotlin отримати та застосувати
↪  останню версію сховища
%useLatestDescriptors
//Включити весь необхідний шаблонний код
↪  Multik до блокнота
%use multik
```

Для початку роботи з бібліотекою multik потрібно створити масив (розмірність від 1 до 4). Декілька стандартних прикладів:

In [2]:
```
val a = mk.ndarray(mk[1, 2, 3])
a
```

Out [2]:
```
[1, 2, 3]
```

In [3]:
```
val b = mk.ndarray(mk[mk[1.5, 2.1, 3.0],
↪  mk[4.0, 5.0, 6.0]])
b
```

Out [3]:
```
[[1.5, 2.1, 3.0],
[4.0, 5.0, 6.0]]
```

Масив, заповнений нулями:

In [4]:
```
mk.zeros<Double>(3, 4)
```

Out [4]:
```
[[0.0, 0.0, 0.0, 0.0],
[0.0, 0.0, 0.0, 0.0],
[0.0, 0.0, 0.0, 0.0]]
```

Тримірний масив x^2:

In [5]:
```
mk.d3array(2, 2, 3) { it * it }
```

Out [5]:
```
[[[0, 1, 4],
[9, 16, 25]],

[[36, 49, 64],
[81, 100, 121]]]
```

Створення масиву в інтервалі $[10, 25]$ з кроком 5:

In [6]:
```
mk.arange<Long>(10, 25, 5)
```

Out [6]:
```
[10, 15, 20]
```

Масив з дев'яти елементі в проміжку $[0, 2]$:

In [7]:
```
mk.linspace<Double>(0, 2, 9)
```

Out [7]:
```
[0.0, 0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75,
↪  2.0]
```

Визначений масив 3 на 3:

In [8]:
```
val e = mk.identity<Double>(3)
e
```

Out [8]:
```
[[1.0, 0.0, 0.0],
[0.0, 1.0, 0.0],
[0.0, 0.0, 1.0]]
```

Також існує можливість роботи з масивами комплексних чисел:

In [9]:
```
mk.d2arrayIndices(3, 3) { i, j ->
↪  ComplexFloat(i, j) }
```

Out [9]:
```
[[0.0+(0.0)i, 0.0+(1.0)i, 0.0+(2.0)i],
[1.0+(0.0)i, 1.0+(1.0)i, 1.0+(2.0)i],
[2.0+(0.0)i, 2.0+(1.0)i, 2.0+(2.0)i]]
```

Звернувши увагу на попереді приклади коду, можна зауважити існування об'єкта `mk`, який є абревіатурою

```
typealias mk = Multik
```

`Multik` – основний об'єкт, через який викликаються всі функції ndarray. Він надає доступ до `ndarray` та створює інтерфейси Math, LinAlg і Statistics. Виклик `Multik` завантажує рушій бібліотеки, через `Multik` Ви можете налаштувати власну реалізацію інтерфейсу.

Властивості масиву. В пакеті доступні такі властивості:

```
a.shape // Розміри масиву
a.size // Розмір масиву
a.dim // Розмір об'єкта
a.dim.d // Кількість вимірів масиву
a.dtype // Тип даних елементів масиву
```

Арифметичні дії

- додавання,
- різниця,
- множення,
- ділення.

Математика масивів. Інші методи лінійної алгебри описано в документації [15].

```
a.sin() // поелементний sin, еквівалент mk.math.sin(a)
a.cos() // поелементний cos, еквівалент mk.math.cos(a)
b.log() // поелементний натуральний логарифм,
        //еквівалент mk.math.log(b)
b.exp() // поелементна експонента, еквівалент
↪  mk.math.exp(b)
d dot e // скалярний добуток, еквівалент
↪  mk.linalg.dot(d, e)
```

Агрегатні функції, доступні в пакеті multik:

```
mk.math.sum() // сума по масиву
mk.math.min() // мінімальні елементи по масиву
mk.math.maxD3(x, axis=0) // максимальне значення
        //масиву вздовж осі 0
mk.math.cumSum(x, axis=1) // кумулятивна сума елементів
mk.stat.mean() // середнє значення
mk.stat.median() // медіана
```

Копіювання масивів відбувається такими методами:

```
val f = a.copy() // створити копію масиву та його даних
val h = b.deepCopy() // створити копію масиву
        //та скопіювати  дані
```

Ітераційні операції, доступні в пакеті multik, коротко можна проілюструвати так:

```
c.filter { it < 3 } // вибрати всі елементи менше 3
b.map { (it * it).toInt() } // повернення квадратів
c.groupNDArrayBy { it % 2 } // згрупувати елементи за
↪  умовою
c.sorted() // сортування елементів
```

Індексування (ітерування) відбувається так:

```
a[2] // вибрати елемент з індексом 2
b[1, 2] // вибрати елемент у рядку 1, стовпці 2
b[1] // вибрати рядок 1
b[0..2, 1] // вибір елементів у рядках 0 і 1 у стовпці 1
b[0..1..1] // вибрати всі елементи в рядку 0
```

також можна використовувати функції `forEach()` та `forEachIndexed()` для поелементного доступу до елементів масиву.

In [10]:
```
 for (el in b) {
        print("$el, ")
}
```

Out [10]:
```
1.5, 2.1, 3.0, 4.0, 5.0, 6.0,
```

для n-мірного:

In [11]:

```
 val q = b.asDNArray()
for (index in q.multiIndices) {
        print("${q[index]}, ")
}
```

Out [11]:

```
 1.5, 2.1, 3.0, 4.0, 5.0, 6.0,
```

Отже, Multik пропонує як багатовимірні структури даних, так і реалізацію математичних операцій над ними. Бібліотека має простий і зрозумілий API та забезпечує оптимізовану продуктивність.

2.2.2. Kotlin-Statistics: математичні та статистичні розширення для Kotlin

Спільнота Kotlin не залишила без уваги такий розділ науки про дані, як математична статистика, і реалізувала бібліотеку kotlin-statistics [16]. Ця бібліотека містить корисні функціональні розширення для проведення статистичних досліджень у ідіоматичний спосіб Kotlin. Іншими словами, користуючись kotlin-statistic можна аналізувати ООП/функціональні дані, не вдаючись до фреймів даних та інших наукових структур даних.

Інструкції зі застосування. Для додавання бібліотеки до проєкту потрібно завантажити файл з репозиторія Maven, відповідна конфігурація

build.gradle.kts
```
dependencies {
        implementation("org.nield:kotlin-statistics:1.2.1")
}
```

Ви також можете використовувати Maven або Gradle з JitPack для безпосереднього створення знімка як залежності.

build.gradle.kts
```
repositories {
    maven("https://jitpack.io")
}
dependencies {
    implementation("com.github.thomasnield:
↪   kotlin-statistics: -SNAPSHOT")
}
```

Основні оператори. Існує чимало операторів функцій розширення, які підтримують числові типи `Int`, `Long`, `Double`, `Float`, `BigDecimal` і `Short` для послідовностей, масивів та ітерованих елементів: `descriptiveStatistics`, `sum()`, `average()`, `min()`, `max()`, `mode()`, `median()`, `range()`, `percentile()`, `variance()`,

`standardDeviation()`, `geometricMean()`, `sumOfSquares()`, `normalize()`, `simpleRegression()`, `kurtosis`, `skewness`.

Ось приклад використання функції розширення `median()` до послідовності подвійних значень:

```
val median = sequenceOf(1.0, 3.0, 5.0).median()
println(median) // prints "3.0"
```

Оператори отримання вибірки даних. Існують також прості, проте потужні оператори `xxxBy()`, які допомагають розділити багато з цих статистичних операторів за певним ключем:

- `countBy()`
- `sumBy()`
- `averageBy()`
- `geometricMeanBy()`
- `minBy()`
- `maxBy()`
- `rangeBy()`
- `varianceBy()`
- `standardDeviationBy()`
- `descriptiveStatisticsBy()`
- `simpleRegressionBy()`

```
//Знайти суми за довжиною імені, використовуючи пари або
// функціональні аргументи
val sumsByLengths = sequence
       .map { it.name.length to it.value }
       .sumBy()
```

```
// Або
val sumsByLengths = sequence
       .sumBy(keySelector = { it.name.length },
              doubleSelector = {it.value} )

println("Sums by lengths: $sumsByLengths")
```

Ці оператори вибірки підтримуються загальною функцією `groupApply()`, яку можна легко використовувати для реалізації інших операторів нарізки. Ви можете розділити кілька полів за допомогою класів даних за допомогою операторів `xxxBy()`, це схоже на використання GROUP BY для кількох полів у SQL. Ви також можете групувати за діапазонами (або відомими в статистиці як «біни» або «гістограма»). Існують спеціальні оператори `bin`, які працюють із числовими діапазонами для `Int`, `Long`, `Double`, `Float` та `BigDecimal`. Також Ви можете згрупувати будь-які <T> елементів у контейнери, що складаються з порівняних діапазонів. Якщо Ви хочете виконати математичне агрегування певної властивості для кожного елемента (замість того, щоб групувати елементи в список для певного контейнера), надайте аргумент `groupOp`, який визначає, як обчислити значення для кожного групування.

Випадковий вибір. Kotlin-Statistics має кілька корисних розширень для випадкової вибірки елементів із `Iterable<T>` або `Sequence<T>`.

- `randomFirst()` – вибирає один випадковий елемент, але видає помилку, якщо елементи не знайдено.
- `randomFirstOrNull()` – вибирає один випадковий елемент, але повертає `null`, якщо елементи не знайдено.
- `random(n: Integer)` – вибирає `n` випадкових елементів.
- `randomDistinct(n: Integer)` – вибір `n` різних випадкових елементів.

Замість того, щоб робити чисту випадкову вибірку, може бути, що Ви захочете, щоб різним значенням типу `T` присвоїли різні ймовірності, а потім Ви захочете відібрати `T` випадково, ураховуючи ці ймовірності. Це може бути корисним для створення симуляцій або стохастичних алгоритмів загалом.

`WeightedCoin` та `WeightedDice` допомагають у цих цілях. `WeightedCoin` приймає значення `trueProbability` від 0,0 до 1,0. Якщо ми задаємо ймовірність 0,80, монета підкидатиметься приблизно у 80% випадків.

```
val riggedCoin = WeightedCoin(trueProbability = .80)

// підкинути монету 100 000 разів і роздрукувати
↪  результат
(1..100000).asSequence().map { riggedCoin.flip() }
        .countBy()
        .also {
            println(it)
        }
```

Ви можете використовувати `WeightedDice` для керування результатами, зіставленими з будь-яким типом `T`. Наприклад, якщо у нас є кубик зі сторонами «A», «B» і «C» з результатами ймовірності 0,11, 0,66 і 0,22, ми можемо ефективно створити `WeightedDice`. Зазвичай у `WeightedDice` Ви, ймовірно, використовуватимете `enum class`, щоб призначити ймовірності для окремих елементів.

Наївний Баєсів класифікатор є сімейством простих «імовірнісних класифікаторів», заснованих на застосуванні теореми Баєса. Застосовують для визначення ймовірності приналежності спостереження (елемента вибірки) до одного з класів за припущення (наївного) незалежності змінних [17].

`NaiveBayesClassifier` виконує просту, але потужну форму машинного навчання. Для заданого набору `T`-елементів Ви можете отримати одну або кілька ознак `F` і пов'язати з категорією `C`. Потім Ви можете перевірити новий набір функцій `F` і передбачити категорію `C`.

Наприклад, ви хочете ідентифікувати електронну пошту як спам / не спам на основі слів у повідомленнях. У цьому випадку можливими категоріями будуть `true` (спам) або `false` (не спам), а кожне слово буде особливістю. У ідіоматичному стилі Kotlin ми можемо взяти простий `List<Email>` і викликати `toNaiveBayesClassifier()`, надати функції вищого порядку для вилучення функцій і категорії, а потім створити модель. Потім ми можемо використовувати цю модель `NaiveBayesClassifier`, щоб передбачити спам нових електронних листів. Якщо Ви хочете додати більше спостережень до Вашої наївної баєсівської моделі, просто виклнчте `addObservation()`, і вона оновить свою ймовірнісну модель.

Кластеризація. У Kotlin-Statistics є кілька алгоритмів кластеризації. Ці алгоритми намагаються згрупувати тісно пов'язані елементи на основі їх близькості на двовимірному графіку. На цей момент доступні чотири методи кластеризації, реалізовані за допомогою *Apache Commons Math* [18].

- **KMeans** має на меті розділити n спостережень на k кластерів так, щоб кожна точка належала до кластеру з найближчим центром.
- **Fuzzy-KMeans** – варіант класичного алгоритму K-Means, з тією основною відмінністю, що окрема точка даних не призначена однозначно одному кластеру. Натомість кожна точка i має набір ваг u_{ij}, які свідчать про ступінь належності до кластера j. Нечіткий варіант не потребує початкових значень для центрів кластерів і, а отже, є більш надійним, хоча й повільнішим, ніж оригінальний алгоритм kMeans.
- **Multi-KMeans** – метаалгоритм, який виконує n прогонів за допомогою KMeans, а потім обирає найкраще кластеризування (тобто таке з найменшою дисперсією відстані серед усіх кластерів) із цих прогонів.
- **DBSCAN** – просторова кластеризація додатків із шумом на основі щільності. DBSCAN знаходить кількість кластe-

рів, починаючи з оціненого розподілу щільності відповідних вузлів. Основними перевагами над KMeans є те, що DBSCAN не потребує визначення початкової кількості кластерів і може знаходити кластери довільної форми.

Порівняння роботи алгоритмів кластеризації наведено на рис. 2.2.

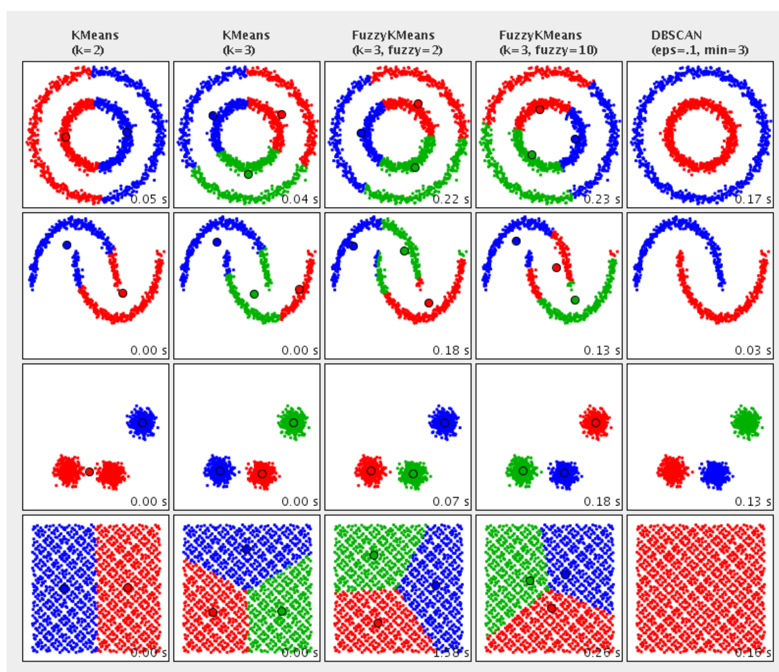


Рис. 2.2. Порівняння доступних алгоритмів кластеризації [18]

У додатку A.1. проведено класифікацію пацієнтів за віком і кількістю лейкоцитів. Зауважте, що аргументи `xSelector` і `ySelector` наразі мають відповідати числовому типу `Double`. Результати кластеризації наведено на рис. 2.3.

Агрегування кількох полів. За допомогою оператора Kotlin `let()` легко взяти колекцію елементів і об'єднати кілька полів у

інший «підсумковий» об'єкт. Так, наприклад, для фільтрації електронної пошти можна агрегувати поля теми та відправника. Ви також можете виконувати різні перетворення для кожного поля, наприклад, розділяти слова та вводити їх у певний регістр перед отриманням розподілу.

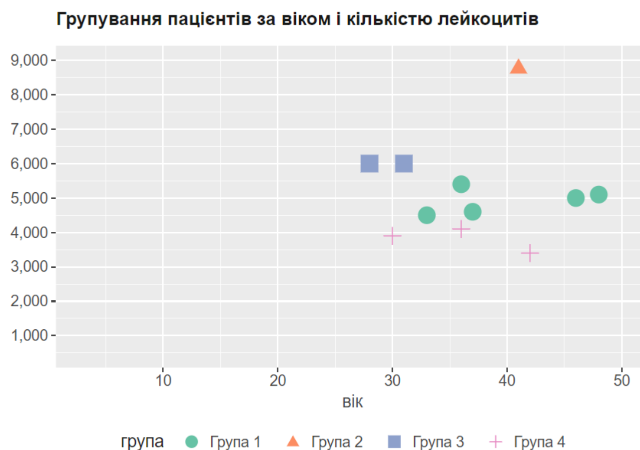


Рис. 2.3. Результат кластеризації, візуалізація Lab-Plot

Повторне використання логіки з функціями розширення. Повернемось до прикладу з додатку A.1.. Скажімо, Ви хочете знайти 1-й, 25-й, 50-й, 75-й і 100-й проценти за статтю. Ми можемо використати функцію розширення Kotlin під назвою `wbccPercentileByGender()`, яка візьме набір пацієнтів і розділить обчислення процентів за статтю. Тоді ми можемо викликати його для п'яти бажаних процентів і запакувати їх у `Map<Double,Map<Gender,Double>>`, як показано нижче:

```
fun
↪  Collection<Patient>.wbccPercentileByGender(percentile:
↪  Double) =
            percentileBy(
                    percentile = percentile,
```

```
                    keySelector = { it.gender },
                    valueSelector = {
↪  it.whiteBloodCellCount }
            )

val percentileQuadrantsByGender = patients.let {
        mapOf(1.0 to it.wbccPercentileByGender(1.0),
                25.0 to it.wbccPercentileByGender(25.0),
                50.0 to it.wbccPercentileByGender(50.0),
                75.0 to it.wbccPercentileByGender(75.0),
                100.0 to
↪  it.wbccPercentileByGender(100.0)
        )
    }

percentileQuadrantsByGender.forEach(::println)
```

Kotlin спрощує повторне використання коду, залишаючись гнучким, тож приділіть трохи часу довіднику Kotlin, щоб відкрити для себе функції, які можна використовувати для вираження бізнес-логіки.

Лінійна регресія також реалізується в бібліотеці kotlin-statistics. Ви можете отримати `SimpleRegression` на `Sequence` або `Iterable`, що емітує `Double` `pairs`, наприклад:

```
fun main(args: Array<String>) {
    val r = sequenceOf(
            1.0 to 3.0,
            2.0 to 6.0,
            3.0 to 9.0
        ).simpleRegression()

    println(r.slope)
    //prints the slope "3.0"
```

Ви також можете вибрати x і y на будь-якому довільному типі `T`.

Загалом kotlin-statistics містить достатньо великий набір інструментів для проведення статистичних досліджень; більш розгорнуту документацію та більше прикладів можна отримати в [16].

2.2.2. Krangl: маніпулювання даними

Krangl ({K}otlin library for data w{rangl}ing) це Kotlin бібліотека, що реалізує граматику маніпулювання даними за допомогою API сучасного функціонального стилю, це помагає фільтрувати, перетворювати, агрегувати та змінювати табличні дані [19] (поставляється з підтримкою JDBC).

Krangl значною мірою натхненний неперевершеним dplyr [20] для R. Krangl написаний у Kotlin та імітує API dplyr, обережно додаючи більше типізованих конструкцій, де це можливо. Крім того, він надає методи переходу між нетиповими та типізованими даними. Отже, що може krangl:

- фільтрувати, перетворювати, агрегувати та змінювати форму табличних даних;
- читати звичайний і стиснутий формат tsv, csv, json або будь-який інший формат із роздільниками, заголовком або без нього з локального чи віддаленого пристрою. Таблиці можуть містити атомарні стовпці (`Int`, `Double`, `Boolean`), а також стовпці об'єктів;
- змінювати форму таблиць із широких на довгі та назад;
- об'єднувати таблиці (ліворуч, праворуч, напів, внутрішнє, зовнішнє);
- виконує статистичний аналіз (середнє значення, мінімум, максимум, медіана, ...).

Щоб почати, просто додайте його як залежність до свого проєкту:

build.gradle.kts

```
repositories {
  mavenCentral()
}
dependencies {
  implementation("com.github.holgerbrandl:krangl:0.18.1")
}
```

Ви також можете використовувати репозитарій JitPack з Maven або Gradle, щоб використати останній варіант як залежність у Вашому проєкті.

build.gradle.kts

```
dependencies {
    implementation("com.github.holgerbrandl:
↪   krangl:-SNAPSHOT")
}
```

Як і у випадку інших бібліотек, радимо почати роботу з krangl в записнику jupyter [1]. Ви можете включити його до свого сценарію записника jupyter за допомогою анотації `%use krangl`.

Модель даних krangl. DataFrame – це «таблична» структура даних, яка містить записи (рядки), кожен з яких складається зі спостережень або вимірювань (стовпці). Отже, зіставляючи це визначення з кодом Kotlin, ми отримуємо основну абстракцію krangl:

```
interface DataFrame {
    val cols: List<DataCol>
}

abstract class DataCol(val name: String) {
    abstract fun values(): Array<*>
}
```

Krangl застосовує модель стовпців, щоб уможливити векторизацію, де це можливо. Реалізації стовпців проходить із використанням нульових типів `String?`, `Int?`, `Double?`, `Boolean?` та `Any?`. Відбувається внутрішня перевірка узгодженості довжини та типу (наприклад, запобігання повторюваних імен стовпців). Також krangl змішує типізовані та нетипізовані дані в табличній структурі, він реалізує API pandas/tidyverse для створення, маніпулювання, зміни форми, комбінування та узагальнення кадрів даних.

Існує декілька можливостей перенесення потрібних Вам даних у krangl. Найпопулярніший – читати з файлів tsv, csv, json, jdbc

тощо.

```
val tornados = DataFrame.readCSV(pathAsStringFileOrUrl)
tornados.writeCSV(File("tornados.txt.gz"))
```

Інакший варіант – створення об'єкта `DataFrame` в коді програми, або в записнику jupyter. Далі розглянемо такий випадок.

Основні методи маніпулювання даними. Отже, включимо потрібний шаблонний код krangl до блокнота

In [1]:
```
%use krangl
```

Створимо кадр даних у пам'яті

In [2]:
```
val df: DataFrame = dataFrameOf(
    "first_name", "last_name", "age",
↪   "weight")(
    "Max", "Doe", 23, 55,
    "Franz", "Smith", 23, 88,
    "Horst", "Keanes", 12, 82
)
```

Після створення (зчитування) Ви можете дослідити дані, користуючись деякими методами. Найпростіший виведе обмежений, у випадку великої таблиці, вміст:

In [3]:
```
df
```

Out [3]:
```
first_name  last_name   age  weight
Max         Doe         23   55
Franz       Smith       23   88
Horst       Keanes      12   82
Shape: 3 x 4.
```

перелік методів значно ширший, наприклад: `head()` – побачити перші п'ять рядків; `tail()` – відповідно, останні; властивість `rows` і метод `rows.elementAt()` дає можливість отримати рядок, індекс якого дорівнює 0; `slice(a..b)` повертає кадр даних, який починається з і включає рядок `a` закінчується та включає рядок `b` підраховує рядки, починаючи з 1, а не з 0. Структуру Ваших даних відображає метод `schema()` . Змінити структуру можна простим додаванням колонки:

In [4]:
```
df.addColumn("salary_category") { 3
↪  }.schema()
```

Out [4]:
```
Name         Type    Values
first_name        [Str]   Max, Franz, Horst
last_name         [Str]   Doe, Smith, Keanes
age               [Int]   23, 23, 12
weight            [Int]   55, 88, 82
salary_category   [Int]   3, 3, 3
DataFrame with 3 observations
```

Ви можете додати кілька стовпців одночасно

```
df.addColumns(
        // виконуючи базову арифметику стовпців
    "age_plus3" to { it["age"] + 3 },
    //створити нові атрибути за допомогою рядкових
↪   операцій, таких як зіставлення, розбиття або
↪   вилучення.
    "initials" to { it["first_name"].map<String> {
↪   it.first() } concat it["last_name"].map<String> {
↪   it.first() } }
)
```

Щоб створити стовпці, які починаються з постійних значень, їх потрібно розширити до статичних стовпців за допомогою `const`

```
df.createColumn("user_id") { const("id") + nrow }
```

krangl використовує «null» як відсутнє значення та надає зручні методи для їх обробки.

Сортування даних за допомогою `sortedBy`**.** Основна функція для впорядкування даних `sortedBy()`, аргумент якої має атрибут `varargs`, наприклад:

In [5]:
```
df.sortedBy("age", "weight")
```

Out [5]:
```
first_name  last_name  age  weight
Franz       Smith      23   88
Max         Doe        23   55
Horst       Keanes     12   82
Shape: 3 x 4.
```

зворотний порядок сортування:

```
df.sortedByDescending("age")
df.sortedBy{ desc("age") }
```

сортувати за спаданням за віком і розв'язувати зв'язки за вагою

```
df.sortedBy({ desc(it["age"]) }, { it["weight"] })
```

сортування з індикатором лямбда

```
df.sortedBy { it["weight"].round() }
```

Підмножина змінних із `select`**.** Зробити вибірку певних полів Ви можете, подібно до мови SQL, методом `select`

In [6]:
```
df.select("last_name", "weight")
```

Out [6]:
```
last_name   weight
Doe         55
Smith       88
Keanes      82
Shape: 3 x 2.
```

вибірка з виключенням деяких полів:

```
df.remove("weight", "age")
```

використання мінімови:

```
df.select { endsWith("name") }
df.select { matches("foo[0-9]") }
```

вибір стовпця функціонального стилю:

```
df.select { it is IntCol }
```

перейменувати стовпці:

```
df.rename("last_name" to "Nachname")
```

Підмножина записів із фільтром. Метод для фільтрації даних у krangl, як не дивно, `filter`. Рядки підмножини з векторизованим фільтром:

In [7]:
```
df.filter { it["age"] eq 23 }
```

Out [7]:
```
first_name last_name age weight
Max        Doe       23  55
Franz      Smith     23  88
Shape: 2 x 4.
```

ще декілька варіантів використання:

```
df.filter { it["weight"] gt 50 }
df.filter({ it["last_name"].isMatching {
↪  startsWith("Do") }})
```

Якщо векторизовані операції неможливі або недоступні, ми також можемо фільтрувати таблиці за рядками, що дає можливість використовувати скалярні оператори

```
df.filterByRow { it["age"] as Int > 5 }
// "round" дні народження :-)
df.filterByRow { (it["age"] as Int).rem(10) == 0 }
```

Узагальніть свої дані за допомогою `summarize`. Обчислення статистики стовпців надає чимало зручних математичних і статистичних методів для часто виконуваних обчислень у числових стовпцях. Зробити прості перехресні таблиці Ви можете так:

In [8]:
```
df.count("age", "last_name")
```

Out [8]:
```
age last_name n
23  Doe       1
23  Smith     1
12  Keanes    1
Shape: 3 x 3.
```

обчислити одну зведену статистику:

```
df.summarize("mean_age") { it["age"].mean(true) }
```

кілька зведених статистичних даних:

In [9]:
```
df.summarize(
    "min_age" to { it["age"].min(true) },
    "max_age" to { it["age"].max(true) }
)
```

Out [9]:
```
min_age  max_age
12.0     23.0
Shape: 1 x 2.
```

Ці методи є логічним аргументом, `removeNA`, який визначає, чи виключати відсутні значення з обчислення. Значення за замовчуванням – `false`, але зазвичай краще встановити `true`.

Виконувати згруповані операції після `groupBy`. За наявності одного або кількох імен стовпців метод `group()` з `DataFrame` створює новий фрейм даних із існуючого, проте організований у набір менших фреймів даних, так звані групи. Кожна група складається з рядків, які мали однакові значення для наведених стовпців.

```
val groupedDf: DataFrame = df.groupBy("age")
```

або надайте кілька атрибутів групування за допомогою змінних

In [10]:
```
val sumDF = groupedDf.summarize(
    "mean_weight" to {
 ↪  it["weight"].mean(removeNA = true) },
    "num_persons" to { nrow }
)
```

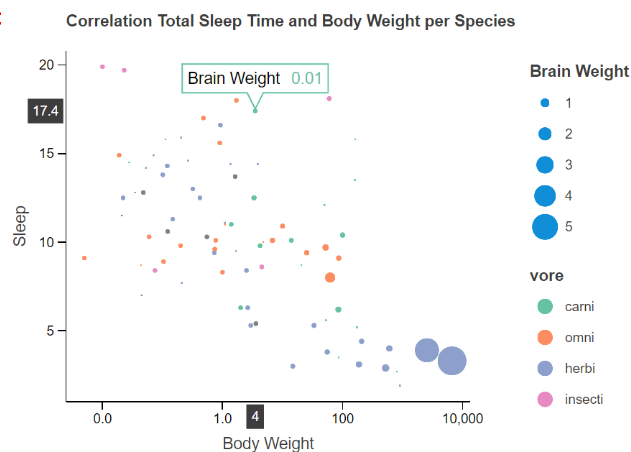
Out [10]:
```
agec mean_weight num_persons
23   71.5        2
12   82.0        1
Shape: 2 x 3.
```

Комбінуючи застосування перелічених методів, Ви можете формувати звіти будь-якої складності та проводити попередній статистичний аналіз. Ще один корисний ресурс для початку роботи з krangl в середовищі jupyter [21].

krangl і графіка. Вище показано деякі базові можливості бібліотеки krangl, однак було б неправильно не звернути уваги на можливості взаємодії з графічними бібліотеками lets-plot та kravis. Для демонстрації скористаємось відомим набором даних sleepData, який містить тривалість сну та ваги для набору видів ссавців. Набір даних містить 83 рядки та 11 змінних. Набір даних поставляється з krangl як приклад, тому немає необхідності завантажувати його з іншого місця. У випадку візуалізації з lab-plot опустимо кроки, потрібні для завантаження бібліотеки та формування вхідних структур даних, які описано в розділі 2.2.2.

In [11]:
```
letsPlot(data) {
        x="Body Weight"; y="Sleep";
        color="vore"; size="Brain Weight"
} + geomPoint(alpha=.7)
+ scaleXLog10("Body Weight") +
ggtitle("Correlation Total Sleep Time and
↪  Body Weight per Species")
```

Out [11]:



Що стосується візуалізації з використанням бібліотеки kravis, рекомендовано авторами, в додатку A.2. наведено програмний код, результати на рис.2.1, 2.4. У прикладі використано вже згадуваний у розділі 2.2.2. набір даних sleepData, що входить до бібліотеки.

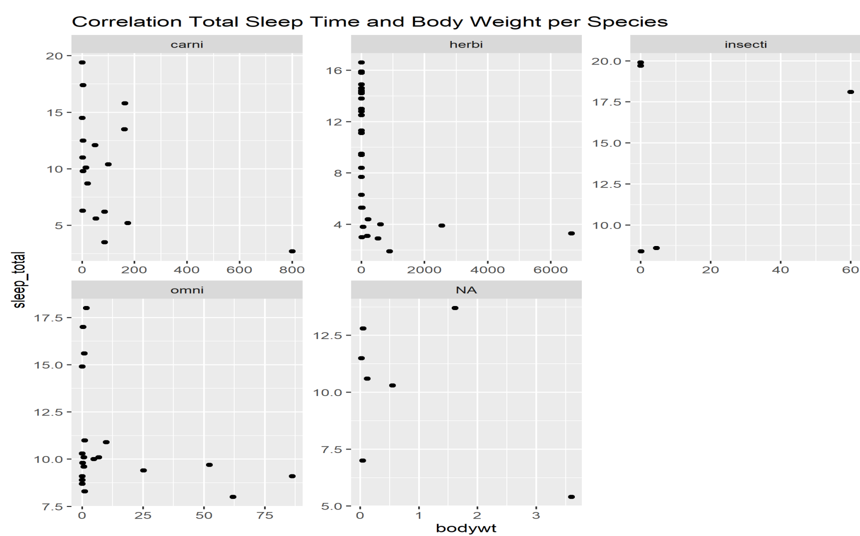


Рис. 2.4. Результат спільного застосування бібліотек kravis та krangl, код програми в додатку A.2..

2.2.2. KotlinDL: API глибинного навчання високого рівня в Kotlin

Спільнота Kotlin не залишила без уваги такий розділ науки про дані, як штучний інтелект, і створила бібліотеку глибинного навчання (Deep Learning). KotlinDL – це високорівневий API глибинного навчання [22], написаний мовою Kotlin і створений за мотивами Keras [23]. Під капотом він використовує TensorFlow Java API [24] та ONNX Runtime API для Java [25]. KotlinDL пропонує прості API для навчання моделей глибинного навчання з нуля, імпорту існуючих моделей Keras і ONNX, а також адаптації попередньо навчених моделей до ваших завдань.

Щоб використовувати всю потужність KotlinDL (включно з ONNX і модулями візуалізації) у вашому проєкті, додайте такі залежності до свого файлу build.gradle.kts:

build.gradle.kts

```
repositories {
  mavenCentral()
}
dependencies {
  implementation("org.jetbrains.kotlinx:kotlin-
↪  deeplearning-api:0.3.0'")
  implementation("org.jetbrains.kotlinx:kotlin-
↪  deeplearning-onnx:0.3.0")
  implementation("org.jetbrains.kotlinx:kotlin-
↪  deeplearning-visualization:0.3.0")
}
```

Додайте лише одну залежність, якщо Вам не потрібні ONNX і візуалізація:

build.gradle.kts

```
  implementation("org.jetbrains.kotlinx:kotlin-
↪  deeplearning-api:0.3.0'")
```

Тренування моделей на центральному процесорі може тривати значний час. Найпоширенішою практикою є запуск обчислень на GPU. Щоб увімкнути навчання нейромережі та обчислення на графічному процесорі (GPU), прочитайте сторінку під-

тримки TensorFlow графічного процесора та встановіть фреймворк CUDA. Зауважте, що підтримуються лише пристрої NVIDIA.

Ви можете працювати з KotlinDL в інтерактивному режимі в Jupyter Notebook з ядром Kotlin. Для цього додайте таку залежність у свій блокнот:

In [1]:
```
@file:DependsOn("org.jetbrains.kotlinx:kotlin-
↪  deeplearning-api:0.3.0")
```

У документації до проєкту [22] Ви знайдете статті, які, сподіваємось, допоможуть Вам спробувати цей фреймворк:

- короткий посібник;
- створення першої нейронної мережі;
- навчання моделі;
- одержання логічних висновків із навченою моделлю;
- імпорт моделі Keras;
- трансфер моделей навчання;
- передача моделей за допомогою функціонального API.

Колекція попередньо тренованих моделей ResNet та MobileNet. Починаючи з релізу 0.2, у Kotlin DL з'являється «зоопарк моделей» (або Model Zoo). По суті, це колекція моделей з вагами, що отримані під час навчання на великій кількості зображень.

Навіщо потрібна така колекція моделей? Справа в тому, що сучасні надточні нейромережі можуть мати сотні шарів та мільйони параметрів, що оновлюються багаторазово впродовж кожної ітерації навчання. Тренування моделей до прийнятного рівня точності (70-80 %) на такому великому датасеті, як ImageNet, може займати сотні та тисячі годин обчислювального часу великого кластера з відеокарт.

Зоопарк моделей дає змогу вам користуватися вже готовими та натренованими моделями (Вам не доведеться тренувати їх з нуля щоразу, коли вони Вам потрібні). Ви можете використовувати таку модель безпосередньо для передбачень. Також Ви можете застосувати її для дотренування частини моделі на невеликій порції вхідних даних – загалом, це дуже поширена техніка під час використання перенесення навчання (Transfer Learning). Таке дотренування може зайняти десятки хвилин на одній відеокарті (або навіть центральному процесорі) замість сотень годин на великому кластері.

Для кожної моделі доступні функції завантаження конфігурації моделі в JSON-форматі та ваги у форматі .h5. Також для кожної моделі можна використовувати спеціальний препроцесинг, який застосовували для навчання на датасеті ImageNet[4].

Розпізнавання об'єктів. Це досить простий термін зі світу глибинного навчання,який має за завдання виявлення екземплярів об'єктів певного класу в зображенні. Для демонстрації можливостей бібліотеки KotlinDL пропонуємо розглянути приклад [27, 28]. Передбачається, що Ви уже знайомі з розпізнаванням зображень, де ідея полягає в тому, щоб розпізнати клас або тип лише одного об'єкта на зображенні без жодних координат для розпізнаного об'єкта.

На відміну від розпізнавання зображень, під час виявлення об'єктів ми намагаємося виявити кілька об'єктів (іноді це може бути значна кількість, наприклад 100 або навіть 1 000) та їх розташування, які зазвичай мають вигляд чотирьох координат прямокутника, що містить виявлений об'єкт. Наприклад, на цьому знімку екрана програми рис. 2.5. показано, як було розпізнано кілька об'єктів і позначено їхні позиції.

Припустимо, ми маємо таке зображення рис. 2.6. Ми бачимо типову вулицю: кілька машин, пішохідний перехід, світлофор і навіть хтось їде по пішохідному переходу на велосипеді.

[4] ImageNet – це база даних зображень, організована відповідно до ієрархії WordNet (наразі лише іменники), у якій кожен вузол ієрархії зображено сотнями й тисячами зображень [26].

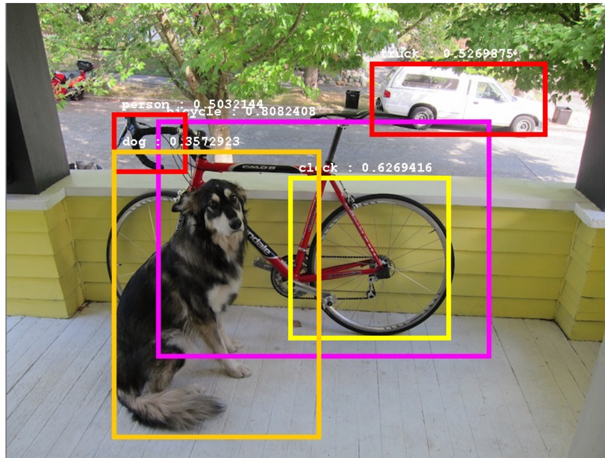


Рис. 2.5. Приклад розпізнавання кількох об'єктів

Рис. 2.6. Вхідне зображення

За допомогою кількох рядків коду ми можемо отримати список виявлених об'єктів, відсортованих за балом або ймовірністю (ступінь упевненості моделі в тому, що певний прямокутник містить об'єкт певного типу).

```
val modelHub = ONNXModelHub(cacheDirectory =
↪  File("cache/pretrainedModels"))

val model =
↪  modelHub.loadPretrainedModel(ONNXModels.ObjectDetection.SSD)

model.use { detectionModel ->
   println(detectionModel)

   val imageFile =
↪  getFileFromResource("detection/image2.jpg")
   val detectedObjects =
↪  detectionModel.detectObjects(imageFile = imageFile,
↪  topK = 20)

   detectedObjects.forEach {
       println("Found ${it.classLabel} with probability
↪  ${it.probability}")
   }
}
```

Цей код надрукує таке:

```
Found car with probability 0.9872914
Found bicycle with probability 0.9547764
Found car with probability 0.93248314
Found person with probability 0.85994
Found person with probability 0.8397419
Found car with probability 0.7488473
Found person with probability 0.49446288
Found person with probability 0.48537987
Found person with probability 0.40268868
Found person with probability 0.3972058
Found person with probability 0.38047826
Found traffic light with probability 0.36501375
...
```

Як Ви можете бачити, API виявлення об'єктів повертає не лише назву класу та оцінку, а й відносні координати зображення, які можна використовувати для малювання прямокутників або рамок навколо виявлених об'єктів.

Схоже, що модель може виявляти об'єкти так само, як це можуть робити наші очі. Ми можемо використовувати Java-бібліотеку Swing, щоб намалювати прямокутники поверх зображення. Це також потребує простої попередньої обробки зображення перед візуалізацією. Малювання прямокутників на зображенні за допомогою API Graphics2D може бути не найкращим підходом, але ми можемо використовувати його як хорошу відправну точку для Вашого наступного дослідження.

Як наслідок маємо таке зображення:

Рис. 2.7. Результат розпізнавання зображення

Проєкт KotlinDL має на меті зробити глибинне навчання простішим для розробників JVM, що суттєво спрощує розгортання моделей глибинного навчання в середовищах JVM. У KotlinDL Ви знайдете прості API як для опису, так і для тренування нейронних мереж.

Контрольні запитання та завдання

1. Перелічіть інтерактивні редактори для роботи з даними, що використовують мову Kotlin.
2. Які особливості інтерактивних редакторів Вам запам'ятались. Напишіть просту програму на Kotlin в інтерактивному редакторі.
3. Які типи графіків можна відобразити, використовуючи бібліотеку Lets-Plot?
4. Перелічить основні компоненти шарів бібліотеки Lets-Plot.
5. Які типи візуалізації підтримує бібліотека Lets-Plot?
6. Яка мова програмування потрібна для роботи бібліотеки kravis?
7. Які розмірності масивів підтримує бібліотека Multik?
8. Наведіть приклад створення масиві у бібліотеці Multik.
9. Назвіть основний об'єкт, через який викликаються всі функції ndarray?
10. Які оператори отримання вибірки даних підтримує бібліотека Kotlin-Statistics?
11. Напишіть програму з використанням алгоритмів кластеризації.
12. Напишіть програму лінійної регресії.
13. Який основний функціонал надає бібліотека маніпулювання даними Krangl?
14. Опишіть модель даних бібліотеки Krangl.
15. Напишіть програму фільтрації даних, використовуючи бібліотеку Krangl.

16. Який основний функціонал надає бібліотека глибинного навчання KotlinDL?

17. Спробуйте запустити приклад розпізнавання об'єктів з власними зображеннями, використовуючи різні моделі.

Список використаних джерел

1. Project Jupyter. URL: https://jupyter.org (дата звернення: 03.02.2023).
2. JetBrains Datalore: A powerful environment for Jupyter notebooks. URL: https://datalore.jetbrains.com (дата звернення: 22.06.2022).
3. Zeppelin / Apache Software Foundation. URL: https://zeppelin.apache.org (дата звернення: 03.12.2022).
4. *Khalusova M.* Lets-Plot, in Kotlin. 17.12.2022. URL: https://blog.jetbrains.com/kotlin/2020/12/lets-plot-in-kotlin (дата звернення: 02.03.2023).
5. Jupyter Notebook Viewer. URL: https://nbviewer.org/github/JetBrains/lets-plot-kotlin/blob/master/docs/guide/user_guide.ipynb (дата звернення: 24.07.2022).
6. *Wickham H.* ggplot2: Elegant Graphics for Data Analysis. 2nd ed. Springer New York, NY, 2016. 260 p. ((Use R!)) DOI: https://doi.org/10.1007/978-0-387-98141-3.
7. An Open-source Plotting Library for Statistical Data. URL: https://lets-plot.org (дата звернення: 02.08.2022).
8. Lets-Plot for Kotlin / Jet Brains. URL: https://github.com/JetBrains/lets-plot-kotlin (дата звернення: 02.04.2023).

9. *Brandl H.*, *Matsuno T.* A {K}otlin g{ra}mmar for data {vis}ualization. URL: https://github.com/holgerbrandl/kravis (дата звернення: 04.08.2022).

10. *deVilla J.* Rserve – Binary R server – RForge.net. URL: https://www.rforge.net/Rserve/doc.html (дата звернення: 22.08.2022).

11. *Brandl H.* Kalasim. URL: https://github.com/holgerbrandl/kalasim/blob/master/simulations/notebooks/kravis_test.ipynb (дата звернення: 14.08.2022).

12. Kotlin Programming Language / Jet Brains. URL: https://kotlinlang.org (дата звернення: 14.08.2022).

13. Multidimensional array library for Kotlin. URL: https://github.com/Kotlin/multik (дата звернення: 16.08.2022).

14. *Khalusova M.* Multik: Multidimensional Arrays in Kotlin. URL: https://blog.jetbrains.com/kotlin/2021/02/multik-multidimensional-arrays-in-kotlin (дата звернення: 22.11.2022).

15. Multik-Core. URL: https://kotlin.github.io/multik/ (дата звернення: 22.08.2022).

16. *Nield T.* Kotlin-statistics: Idiomatic statistical operators for Kotlin. URL: https://github.com/thomasnield/kotlin-statistics (дата звернення: 29.07.2022).

17. *Rish I.* An empirical study of the naive Bayes classifier // IJCAI 2001 Workshop on Empirical Methods in Artificial Intelligence. 2001. P. 41–46.

18. Math – The Commons Math User Guide - Machine Learning / Apache Software Foundation. URL: https://commons.apache.org/proper/commons-math/userguide/ml.html (дата звернення: 22.08.2022).

19. *Brandl H.* A {K}otlin library for data w{rangl}ing. URL: https://github.com/holgerbrandl/krangl (дата звернення: 11.07.2022).

20. A Grammar of Data Manipulation. URL: https://dplyr.tidyverse.org (дата звернення: 11.07.2022).

21. Beginning Data Science with Jupyter Notebook and Kotlin. URL: https://www.raywenderlich.com/27470499-beginning-data-science-with-jupyter-notebook-and-kotlin (дата звернення: 22.08.2022).
22. KotlinDL: High-level Deep Learning API in Kotlin. URL: https://github.com/Kotlin/kotlindl (дата звернення: 14.08.2022).
23. Keras: the Python deep learning API. URL: https://keras.io/ (дата звернення: 18.08.2022).
24. TensorFlow for Java. URL: https://www.tensorflow.org/jvm (дата звернення: 18.08.2022).
25. Open Neural Network Exchange. URL: https://onnx.ai/ (дата звернення: 18.08.2022).
26. ImageNet. URL: https://image-net.org/ (дата звернення: 15.09.2022).
27. *Zinoviev A.* Ktor KotlinDL Object Detection examples. URL: https://github.com/zaleslaw/Ktor-KotlinDL-Object-Detection-examples (дата звернення: 28.09.2022).
28. *Zinoviev A.* Object Detection with KotlinDL and Ktor. 31.01.2022. URL: https://blog.jetbrains.com/kotlin/2022/01/object-detection-with-kotlindl-and-ktor/ (дата звернення: 28.09.2022).

Додаток А.

Приклади програм

А.1. Використання бібліотек Kotlin-Statistics та Lab-Plot

build.gradle.kts

```
plugins {
    kotlin("jvm") version "1.7.0"
    application
}
val lets_plot_version = "2.3.0"
val lets_plot_kotlin_version = "3.2.0"
val slf4j_version = "1.7.32"

repositories {
    mavenCentral()
    maven("https://jitpack.io")
}
dependencies {
    implementation("com.github.thomasnield:
↪   kotlin-statistics: -SNAPSHOT")
    implementation("org.jetbrains.lets-plot:
↪   lets-plot-common: $lets_plot_version")
    implementation("org.jetbrains.lets-plot:
↪   lets-plot-kotlin-jvm: $lets_plot_kotlin_version")
    implementation("org.slf4j:slf4j-simple:
↪   $slf4j_version")
```

```
}
application {
    mainClass.set("MainKt")
}
```

```
import org.nield.kotlinstatistics.multiKMeansCluster
import jetbrains.letsPlot.export.ggsave
import jetbrains.letsPlot.geom.geomDensity
import jetbrains.letsPlot.geom.geomPoint
import jetbrains.letsPlot.label.ggtitle
import jetbrains.letsPlot.letsPlot
import jetbrains.letsPlot.scale.scaleXDiscrete
import jetbrains.letsPlot.scale.scaleYDiscrete
import jetbrains.letsPlot.theme
import jetbrains.letsPlot.themeGrey
import java.time.temporal.ChronoUnit
import java.time.LocalDate

fun main(args: Array<String>) {

    /*
     Групувати пацієнтів за віком і кількістю лейкоцитів
     */
    val k = 4
    val clusters =  patients.multiKMeansCluster(
        k ,
        maxIterations = 10000,
        trialCount = 50,
        xSelector = { it.age.toDouble() },
        ySelector = { it.whiteBloodCellCount.toDouble()
↪  }
    )

    // роздрукувати кластери
    clusters.forEachIndexed { k, item ->
```

```
30          println("CENTROID: $k")
31          item.points.forEach {
32              println("\t$it")
33          }
34      }
35
36      /*
37       Відобразити з Lets-Plot
38       */
39
40      // заповнити дані
41      val ageList = mutableListOf<Int>()
42      val wbccList = mutableListOf<Int>()
43      val grpList =  mutableListOf<String>()
44      clusters.forEachIndexed { k, item ->
45          item.points.forEach {patient ->
46              ageList.add(patient.age.toInt())
47              wbccList.add(patient.whiteBloodCellCount)
48              grpList.add("Група ${k+1}")
49          }
50      }
51
52      val data = mapOf<String, Any>(
53          "група" to grpList,
54          "вік" to ageList,
55          "wbcc" to wbccList
56
57      )
58
59      var p = letsPlot(data){
60          x = "вік"
61          y = "wbcc"
62          color ="група"
63          shape = "група"
64      } + geomPoint(
65          size = 7.0,
```

```
66      )
67      p += ggtitle("Групування пацієнтів за віком і
    ↪   кількістю лейкоцитів")
68      p += themeGrey()
    ↪   //сіра тема
69      p += theme().legendPositionBottom()              //
    ↪   легенда вниз
70      p += scaleXDiscrete(limits = listOf(3,50))       //
    ↪   Осі X, Y
71      p += scaleYDiscrete(name = "", limits =
    ↪   listOf(500,9_000))
72
73      // записати графік, директорія за замовчуванням
    ↪   lets-plot-images
74      ggsave(p,"stat.html")
75  }
76
77  data class Patient(val firstName: String,
78                     val lastName: String,
79                     val gender: Gender,
80                     val birthday: LocalDate,
81                     val whiteBloodCellCount: Int)  {
82
83      val age = ChronoUnit.YEARS.between(birthday,
    ↪   LocalDate.now())
84  }
85
86  val patients = listOf(
87      Patient("John", "Simone", Gender.MALE,
    ↪   LocalDate.of(1989, 1, 7), 4500),
88      Patient("Sarah", "Marley", Gender.FEMALE,
    ↪   LocalDate.of(1970, 2, 5), 6700),
89      Patient("Jessica", "Arnold", Gender.FEMALE,
    ↪   LocalDate.of(1980, 3, 9), 3400),
90      Patient("Sam", "Beasley", Gender.MALE,
    ↪   LocalDate.of(1981, 4, 17), 8800),
```

```
91      Patient("Dan", "Forney", Gender.MALE,
    ↪   LocalDate.of(1985, 9, 13), 5400),
92      Patient("Lauren", "Michaels", Gender.FEMALE,
    ↪   LocalDate.of(1975, 8, 21), 5000),
93      Patient("Michael", "Erlich", Gender.MALE,
    ↪   LocalDate.of(1985, 12, 17), 4100),
94      Patient("Jason", "Miles", Gender.MALE,
    ↪   LocalDate.of(1991, 11, 1), 3900),
95      Patient("Rebekah", "Earley", Gender.FEMALE,
    ↪   LocalDate.of(1985, 2, 18), 4600),
96      Patient("James", "Larson", Gender.MALE,
    ↪   LocalDate.of(1974, 4, 10), 5100),
97      Patient("Dan", "Ulrech", Gender.MALE,
    ↪   LocalDate.of(1991, 7, 11), 6000),
98      Patient("Heather", "Eisner", Gender.FEMALE,
    ↪   LocalDate.of(1994, 3, 6), 6000),
99      Patient("Jasper", "Martin", Gender.MALE,
    ↪   LocalDate.of(1971, 7, 1), 6000)
100 )
101
102 enum class Gender {
103     MALE,
104     FEMALE
105 }
```

А.2. Використання бібліотек Krangl та Kravis

build.gradle.kts

```
1 plugins {
2     kotlin("jvm") version "1.7.0"
3     application
4 }
5
6 repositories {
7     mavenCentral()
```

```
 8      maven("https://jitpack.io")
 9  }
10
11  val krangl_version = "0.18.1"
12  val kravis_version = "0.8.5"
13  dependencies {
14      implementation("com.github.holgerbrandl:
    ↪   krangl:$krangl_version")
15      implementation("com.github.holgerbrandl:
    ↪   kravis:$kravis_version")
16     // testImplementation(kotlin("test"))
17  }
18
19
20  application {
21      mainClass.set("MainKt")
22  }
```

```
 1  import krangl.*
 2  import kravis.*
 3  import kravis.render.LocalR
 4  import kotlin.io.path.Path
 5
 6  fun main(args: Array<String>) {
 7
 8      // sleepData входить до складу Krangl
 9      // як альтернатива  завантажте інші дані
10      sleepData.print()
11
12      // виберіть стовпці, які вас цікавлять
13      val slimSleep = sleepData.select("name",
    ↪   "sleep_total").print()
14
15      // негативне виділення (відоме як видалення
    ↪   стовпців)
16      sleepData.remove("conservation").print()
17
```

```
18      // виберіть діапазон
19      sleepData.select{ range("name", "order")}.print()
20
21      // Виберіть усі стовпці, які починаються з рядка
    ↪   символів "sl" разом зі стовпцем "name"
22      sleepData.select({listOf("name")}, {
    ↪   startsWith("sl")}).print()
23
24      //
25      // Фільтрувати рядки за допомогою`filter`
26      //
27
28      // знайдіть тварин, які сплять більше 16 годин
29      sleepData.filter { it["sleep_total"] gt 16}.print()
30      // це є скороченням для:
31      sleepData.filter { it["sleep_total"].greaterThan(16)
    ↪   }
32
33      //
34      // Kravis графіка
35      //
36
37      SessionPrefs.RENDER_BACKEND = LocalR()
38
39      //  вивести графік у файл
40      sleepData.plot( x="bodywt", y="sleep_total")
41          .geomPoint()
42          .facetWrap("vore", scales=FacetScales.free)
43          .title("Correlation Total Sleep Time and Body
    ↪   Weight per Species")
44          .save(Path("sleepData1.png"))
45
46      sleepData
47          .addColumn("rem_proportion") { it["sleep_rem"] /
    ↪   it["sleep_total"] }
48          //  вивести графік у файл
```

```
49          .plot(x = "sleep_total", y = "rem_proportion",
    ↪  color = "vore", size = "brainwt")
50          .geomPoint(alpha = 0.7)
51          .xLabel("Sleep [h]")
52          .yLabel("")
53          .guides(size = LegendType.none)
54          .title("Correlation between dream and total
    ↪  sleep time")
55          .save(Path("sleepData2.png"))
56  }
57
```

Предметний покажчик

abstract, 36 39
also, 57
Any, 31
Apache Zeppelin, 86
apply, 56
args, 9
Array, 17 64
Array<String>, 9
arrayOf, 17 64
arrayOfNulls, 64
associate, 67
associateBy, 67
associateWith, 67
async, 79
await, 79

Boolean, 16
BooleanArray, 17
break, 21
by, 45
Byte, 14
ByteArray, 17

Char, 16
CharArray, 17
chunked, 70
class, 30
Collection, 57
companion, 32
constructor, 30
continue, 20
copy, 40
Coroutines, 77
CoroutineScope, 77
count, 74

data, 40
Datalore, 85
delay, 78
do-while, 20
Double, 15
DoubleArray, 17
drop, 70

else, 19
emptyArray, 64
enum, 41
equals, 40 52

false, 16
filter, 68
find, 71
findLast, 71
flatMap, 68
flatten, 68
Float, 15
FloatArray, 17
fold, 74

- for, 12
- forEach, 25 27
- forEachIndexed, 27
- fun, 9, 10 21

- ggplot, 92
- ggplot2, 92 104 107
- GlobalScope, 77

- hashCode, 40

- if, 10 18
- if-else, 10 19
- in, 44
- Indexed, 75
- infix, 22
- init, 31
- inline, 27
- inner, 42
- Int, 14
- IntArray, 17
- Intellij Idea, 7
- interface, 38
- internal, 37
- IntProgression, 62
- IntRange, 62
- is, 50
- Iterable, 60
- iterator, 20
- Iterator<>, 20

- Job, 80
- join, 80
- Jupyter Notebook, 82

- kotlin-statistics, 116
    - average, 117
    - DBSCAN, 120
    - descriptiveStatistics, 117
    - geometricMean, 117
    - KMeans, 120
        - Fuzzy, 120
        - Multi, 120
    - kurtosis, 117
    - let, 121
    - max, 117
    - median, 117
    - min, 117
    - mode, 117
    - normalize, 117
    - percentile, 117
    - range, 117
    - simpleRegression, 117
    - skewness, 117
    - standardDeviation, 117
    - sum, 117
    - sumOfSquares, 117
    - variance, 117
    - xxxBy, 117
    - Баєс, 119
- KotlinDL, 136
- krangl, 125
    - addColumn, 128
    - count, 131
    - DataCol, 126
    - DataFrame, 126, 127 132
    - elementAt, 128
    - filter, 130
    - groupBy, 132
    - head, 128
    - max, 132
    - min, 132
    - rename, 130
    - schema, 128

- select, 129
- slice, 128
- sortedBy, 129
- sortedByDescending, 129
- summarize, 131
- tail, 128
- varargs, 129

kravis, 104

lateinit, 34
lazy, 46
let, 55
lets-plot, 91
- addPlot, 101
- coordFixed, 99
- geomBar, 99
- geomDensity, 94
- geomDensity2D, 96
- geomErrorBar, 95
- geomHistogram, 95
- geomPoint, 96
- geomPolygon, 97
- geomTile, 97
- GGBunch, 101
- ggsave, 103
- ggsize, 101
- guideColorbar, 100
- guideLegend, 100
- Legend, 100
- letsPlot, 92 94
- Sampling, 100
- samplingNone, 100
- scaleColorDiscrete, 100
- scaleFillContinuous, 99
- scaleFillHue, 97
- Scales, 99
- show, 104
- statSmooth, 96
- useLatestDescriptors, 93
- Легенда, 100

List, 14 57
listOf, 12, 13
Long, 14
LongArray, 17
LongProgression, 62
LongRange, 62

Map, 57, 58
map, 66
mapOf, 93
maxByOrNull, 73
maxOrNull, 73
maxWithOrNull, 73
minByOrNull, 73
minOrNull, 73
minWithOrNull, 73
Multik, 109
- arange, 111
- ComplexFloat, 112
- copy, 114
- cos, 113
- d2arrayIndices, 112
- d3array, 111
- deepCopy, 114
- dim, 112
- dot, 113
- dtype, 112
- exp, 113
- filter, 114
- forEach, 114
- forEachIndexed, 114
- groupNDArrayBy, 114
- identity, 111
- linspace, 111

log, 113
map, 114
math.cumSum, 113
math.maxD3, 113
math.min, 113
math.sum, 113
mk, 112
multiIndices, 115
ndarray, 110
shape, 112
sin, 113
size, 112
sorted, 114
stat.mean, 113
stat.median, 113
zeros, 111
MutableList, 14
mutableListOf, 13

null, 14
NullPointerException, 14

object, 33
open, 35
operator, 53
out, 44
override, 36 45

package, 37
println, 9
private, 37
protected, 37
public, 37

reduce, 74
return, 25 27
Right, 75
Rserve, 105
run, 55
runBlocking, 78
runningFold, 75
runningReduce, 75

safe-call, 49
sealed, 41
Sequence, 62
SessionPrefs, 104 107
Set, 57, 58
Short, 14
ShortArray, 17
slice, 70
sorted, 73
sortedDescending, 73
star-projection, 48
String, 16
super, 36
suspend, 78

take, 70
this, 26 30 52
Thread, 75
thread, 75
toString, 40
true, 16
try-catch, 11

UByte, 15
UInt, 15
ULong, 15
Unit, 10 22
unzip, 67
UShort, 15

val, 9
var, 9
vararg, 28

when, 10 18
where, 43
while, 13
windowed, 70
with, 56

zip, 67

Навчальне видання

СТАХІРА Роман Йосипович
КУЛИК Петро Русланович
ШУВАР Роман Ярославович

Kotlin для роботи з даними

Навчальний посібник

Львівський національний університет
імені Івана Франка
вул. Університетська 1, м. Львів, 79000