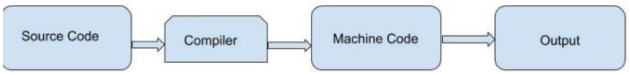
Control questions

1. What is an interpreter?

Інтерпретатор — це програмне забезпечення, яке виконує інструкції кодування безпосередньо, без необхідності їх попередньої компіляції у машинний код. Іншими словами, інтерпретатор аналізує і виконує код у режимі реального часу. Виконує рядок за рядком.

Мови: Python , Javascript, Ruby, PHP



How Compiler Works

How Interpreter Works

3. There are four types of interpreters:

- Bytecode Interpreters
- Threaded code Interpreters
- Abstract Syntax Tree Interpreters
- Self Interpreters

ПЕРЕВАГИ :

1) Інтерпретатори дозволяють запускати код на різних платформах без необхідності змінювати його, оскільки інтерпретатор адаптує виконання до конкретної платформи.
2) Він не створює жодного проміжного об'єктного коду. Отже, це ефективна пам'ять. Він зупиняє компіляцію, якщо виникає будь-яка помилка

НЕДОЛІКИ:

1) Порівняно менше часу займає аналіз вихідного коду. Іншими словами, час компіляції менше. Однак загальний час виконання більше.

   У традиційній компіляції весь код компілюється перед виконанням, що може займати значний час, особливо для великих програм. У Python компіляція у байт-код відбувається під час запуску програми, що дозволяє швидко тестувати і змінювати код.

   Машинний код специфічний для конкретного процесора, тому виконувані файли часто не є портативними між різними платформами. Байт-код інтерпретованих мов, таких як Python, є портативним і може виконуватися на будь-якій платформі, де є відповідний інтерпретатор.

```python
x = 10
if x > 5:
    print("x is greater than 5")
```

1) python example.py - Запуск інтерпретатора:
2) Інтерпретатор читає файл example.py рядок за рядком.
3) Лексичний аналізатор розбиває текст на токени (ключові слова, ідентифікатори, оператори тощо).

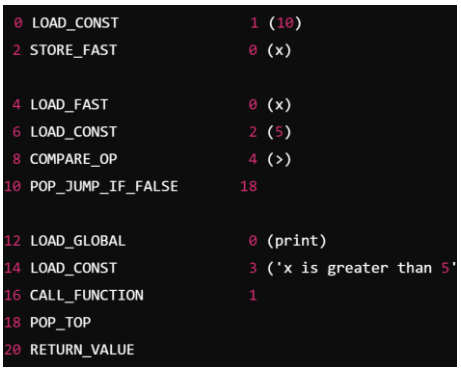
> Приклад токенів для нашого коду:
> - IDENTIFIER(x)
> - ASSIGNMENT_OPERATOR(=)
> - NUMBER(10)
> - KEYWORD(if)
> - IDENTIFIER(x)
> - COMPARISON_OPERATOR(>)
> - NUMBER(5)
> - COLON(:)
> - INDENT
> - IDENTIFIER(print)
> - LEFT_PARENTHESIS(()
> - STRING_LITERAL("x is greater than 5")
> - RIGHT_PARENTHESIS())
> - DEDENT

4) Синтаксичний аналіз (Parsing): Лексичні токени передаються в синтаксичний аналізатор, який будує абстрактне синтаксичне дерево (AST).

```
Assign
  Name(x)
  Constant(10)
If
  Compare
    Name(x)
    Gt
    Constant(5)
  Body
    Expr
      Call
        Name(print)
        Constant("x is greater than 5")
```

5)Побудова байт-коду:

На основі AST інтерпретатор створює байт-код. Байт-код — це низькорівневий проміжний представник коду, який є незалежним від архітектури процесора.



6)Виконання байт-коду (інтерпретація): Віртуальна машина Python (PVM) виконує байт-код інструкція за інструкцією. PVM бере байт-код і інтерпретує його, виконуючи відповідні операції на вашому комп'ютері.

> Завантажує константу 10 і присвоює її змінній x.
> Перевіряє, чи x більше 5.
> Якщо умова істинна, викликає функцію print з аргументом "x is greater than 5".

7)Вивід результату

В результаті виконання коду, інтерпретатор виводить на екран: x is greater than 5.

2. What is a declarative programming language?

   Декларативна мова програмування — це тип мови програмування, де програміст визначає **що** має бути зроблено, а не **як** це має бути зроблено.

Це підхід до програмування, за якого пишуть код , який описує, що вони хочуть робити та яких результатів вони хочуть досягти, а не те, як вони досягнуть результатів. Робота компілятора програми полягає в тому , щоб зрозуміти, як це зробити.

Наприклад, коли ви сідаєте в таксі, ви заявляєте водієві, куди хочете поїхати. Ви не скажете йому, як туди дістатися, надавши покрокові вказівки. Покрокові вказівки схожі на імперативне програмування.

```
SELECT name, age FROM users WHERE age > 30;
```

- Код, написаний декларативно, зазвичай легше читати і розуміти, оскільки він близький до опису бізнес-логіки або кінцевого результату.
- Оскільки програміст фокусується на результаті, а не на кроках виконання, менше шансів допустити помилки в алгоритмах.

Декларативне програмування дозволяє компілятору приймати (як) рішення, тоді як імперативне програмування — ні. Крім того, він не містить змінних змінних, тоді як імперативне програмування включає.

Програмування обмежень/Логічне програмування/Логічне програмування обмежень

Деякі приклади декларативних мов програмування: SQL, HTML, CSS, Prolog, Haskell.

3. What is the functional programming paradigm, and what are its key principles?

Функціональна парадигма програмування - це підхід до програмування, який розглядає обчислення як оцінку математичних функцій і уникає зміни стану та побічних ефектів.

Цей тип зосереджений на декларативному стилі програмування, де акцент робиться на тому, **що** обчислити, а не **як** обчислити. це парадигма створення комп'ютерних програм за допомогою виразів і функцій без зміни стану та даних.

Дотримуючись цих обмежень, функціональне програмування прагне написати код, який є зрозумілішим і більш стійким до помилок. Це досягається шляхом уникнення використання операторів керування потоком
(for, while, break, continue, goto), які ускладнюють дотримання коду. Крім того, функціональне програмування вимагає від нас написання чистих, детермінованих функцій, які мають меншу ймовірність помилок.

> **Чисті функції (Pure Functions):** - Чиста функція завжди повертає один і той же результат для одних і тих же вхідних значень і не має побічних ефектів (тобто не змінює жодних зовнішніх станів).
> Нечисті функції використовують глобальні змінні при обрахуванні
> У функціональному програмуванні змінні є незмінними. Замість зміни значень існуючих змінних створюються нові змінні.
> **Функції вищого порядку (Higher-Order Function) -** Функції вищого порядку - це функції, які приймають інші функції як аргументи або повертають їх як результат
> Immutability: Data structures and variables are not modified once created. – незмінність
> Pure Functions: Functions that produce the same output for the same input, with no side effects on other program parts, the function doesn't modify program's state.

Higher-Order Functions: Functions can take other functions as arguments or return them. Функції вищого порядку
Referential Transparency: - референціальна прозорість (referential transparency) — це властивість виразів у програмуванні, який завжди повертає одне й те саме значення при однакових вхідних даних, і він не має побічних ефектів.

```
def add(a, b):
    return a + b
```

Функція add є референціально прозорою, оскільки для одних і тих же значень a і b вона завжди повертає один і той же результат. Наприклад, add(2, 3) завжди повертає 5, незалежно від контексту.

Каррі

Каррування означає розбиття функції, яка приймає кілька аргументів, на один або декілька рівнів функцій вищого порядку.

Візьмемо функцію `add`.

```
const add = (a, b) => a + b;
```

Коли ми збираємось змінити його, ми переписуємо його, розподіляючи аргументи на кілька рівнів наступним чином.

```
const add = a => {
        return b => {
                return a + b;
        };
};
add(3)(4);
// 7
```

4. Explain the concept of immutability in functional programming and its benefits.

Функціональне програмування бере свій початок у математичних функціях. У реальному світі об’єкти дійсно змінюють свій стан, але в математиці об’єкти не змінюють свого стану.

Як тільки змінній присвоєно значення, це значення не можна змінити протягом усього періоду її існування. Ця послідовність усуває несподівані зміни, роблячи ваш код більш передбачуваним і легшим для міркування. Відсутність змінюваного стану спрощує пошук і виправлення помилок.

Незмінність ідеально узгоджується з концепцією чистих функцій – функцій, які створюють той самий вихід для того самого входу, без побічних ефектів. Чисті

функції легше тестувати, налагоджувати та підтримувати, оскільки вони не змінюють дані поза межами своєї області.

Незмінність спрощує програмування, роблячи код більш передбачуваним, легким для розуміння, безпечним для багатопоточності і полегшує налагодження та тестування. Незмінні об'єкти не змінюються після створення, що дозволяє уникати багатьох помилок, пов'язаних зі зміною стану об'єктів.

5. How can you create pure functions in Python/Haskell/your language?

- Функція завжди повертає однаковий результат для одних і тих самих вхідних значень.
- Функція не має побічних ефектів (не змінює зовнішній стан).

```python
def add(a, b):
    return a + b

def multiply(a, b):
    return a * b

# Виклики функцій
print(add(3, 5))          # Завжди повертає 8
print(multiply(2, 4))     # Завжди повертає 8
```

У Haskell всі функції за замовчуванням є чистими, якщо не використовуються специфічні засоби для роботи з побічними ефектами, такі як монади.

6. What are higher-order functions?

Функції вищого порядку (higher-order functions) — це функції, які можуть приймати інші функції як аргументи або повертати функції як результати. Ця концепція є однією з ключових у функціональному програмуванні, оскільки вона дозволяє створювати більш гнучкий і модульний код.

```python
def apply_function(func, value):
    return func(value)

def square(x):
    return x * x

def cube(x):
    return x * x * x

# Використання функції вищого порядку
print(apply_function(square, 5))  # Повертає 25
print(apply_function(cube, 3))    # Повертає 27
```

```python
def outer_function(x):
    def inner_function(y):
        return x + y
    return inner_function

add_five = outer_function(5)
print(add_five(3))  # Повертає 8
```

7. Describe the role of I/O operations in functional programming and how they are typically handled.

I/O операції (input/output operations) в контексті функціонального програмування відносяться до будь-яких дій, які залучають взаємодію програми із зовнішнім світом, наприклад:

- Читання даних з файлу або запис у файл.
- Введення даних користувачем через клавіатуру.
- Вивід даних на екран.
- Надсилання запитів по мережі або отримання відповідей.

У функціональному програмуванні ці операції є викликом, оскільки вони порушують основний принцип цього підходу - **референціальну прозорість**, що означає відсутність побічних ефектів і передбачуваність результатів функцій.

У Haskell введення-виведення обробляється через монаду IO, щоб зберегти чистоту функцій і референціальну прозорість .Монада — це потужний інструмент у функціональному програмуванні, який допомагає обробляти обчислення з побічними ефектами, зберігаючи основні принципи

8. Compare and contrast interpreted languages with compiled languages.

S. NO.	Скомпільована мова	Інтерпретована мова
1	Скомпільована мова має принаймні два рівні, щоб перейти від вихідного коду до виконання.	Інтерпретована мова виконує один крок, щоб перейти від вихідного коду до виконання.
2	Скомпільована мова перетворюється на машинний код, щоб процесор міг його виконати.	Інтерпретована мова — це мова, у якій реалізації виконують інструкції безпосередньо без попередньої компіляції програми на машинну мову.
4	Зкомпільовані програми працюють швидше, ніж інтерпретовані програми.	Інтерпретовані програми працюють повільніше, ніж скомпільована програма.
5	У скомпільованій мові код може виконуватися ЦП.	У інтерпретованих мовах програма не може бути скомпільована, вона інтерпретується.
6	Ця мова забезпечує кращу продуктивність.	Ця мова забезпечує повільнішу продуктивність.

Інтерпретовані мови :

1) Код виконується "на льоту", без попередньої компіляції.
2) Легше вносити зміни та миттєво їх тестувати, оскільки немає необхідності повторної компіляції коду.
3) Легше налагоджувати та знаходити помилки, оскільки інтерпретатор зазвичай надає детальну інформацію про помилки.
4) Вихідний код може виконуватися на будь-якій платформі, де доступний відповідний інтерпретатор.
5) Виконання інтерпретованого коду вимагає більше пам'яті та процесорного часу.

> ### Внесення змін в код
>
> 1. **Редагування коду:**
>   - Ви змінюєте вихідний код у текстовому редакторі або інтегрованому середовищі розробки (IDE).
> 2. **Запуск інтерпретатора:**
>   - Інтерпретатор зчитує змінений код і виконує його безпосередньо.
>   - Немає необхідності компілювати код у машинний код перед виконанням.