DS-210: Programming for Data Science

Lecture 14: Overview of programming languages.

Comment on Homework 4, Questions 3 & 4

Various levels

- Native code
- Assembler
    - pros: as fine control as in native code
    - cons: not portable

Various levels

- Native code
- Assembler
    - pros: as fine control as in native code
    - cons: not portable
- High level
    - various levels of closeness to the architecture: from C to Prolog
    - efficiency:
        - varies
        - could optimize better
    - pros:
        - very portable
        - easier to build large projects
    - cons:
        - some languages are resource–efficient

Interpreted vs. compiled

Interpreted:

- An application (*interpreter*) reads commands one by one and executes them.
- One step process to run an application:
    - `python hello.py`

Interpreted vs. compiled

Interpreted:

- An application (*interpreter*) reads commands one by one and executes them.
- One step process to run an application:
    - `python hello.py`

("Fully") Compiled:

- Translated to native code by *compiler*
- Usually more efficient
- Two steps to execute:
    1. Compile (Rust: `rustc hello.rs`)
    2. Run (Rust: `./hello`)

Compiled to intermediate format:

- Example: Java
    - Portable intermediate format
    - Needs another application, Java virtual machine, that knows how to interpret it

Type checking: static vs. dynamic

Dynamic (e.g., Python):

- checks if an object can be used for specific operation during runtime
- pros:
  - don't have to specify the type of object
  - procedures can work for various types
  - faster or no compilation
- cons:
  - slower at runtime
  - problems are detected late

```
In [1]: def add(x,y):
            return x + y

        print(add(2,2))
        print(add("a","b"))
        print(add(2,"b"))
```

```
4
ab

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-fc0ed4de2672> in <module>
      4 print(add(2,2))
      5 print(add("a","b"))
----> 6 print(add(2,"b"))

<ipython-input-1-fc0ed4de2672> in add(x, y)
      1 def add(x,y):
----> 2     return x + y
      3
      4 print(add(2,2))
      5 print(add("a","b"))

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Type checking: static vs. dynamic

Static (e.g, C++, Rust, OCaml, Java):

- checks if types of objects are as desired
- pros:
    - faster at runtime
    - type mismatch detected early
- cons:
    - often need to be explicit with the type
    - making procedures generic may be difficult
    - potentially slower compilation

Type checking: static vs. dynamic

Static (e.g, C++, Rust, OCaml, Java):

- checks if types of objects are as desired
- pros:
    - faster at runtime
    - type mismatch detected early
- cons:
    - often need to be explicit with the type
    - making procedures generic may be difficult
    - potentially slower compilation

C++:

```
int add(int x, int y) {
    return x + y;
}
```

Rust:

```
fn add(x:i32, y:i32) -> i32 {
    x + y
}
```

Type checking: static vs. dynamic

Note: some languages are smart and you don't have to always specify types (e.g., OCaml, Rust)

Rust:

```
let x : i32 = 7;
let y = 3;
let z = x * y;
```

Various programming paradigms

- Imperative
- Functional
- Objective
- Declarative / programming in logic

Memory management: manual vs. garbage collection

Manual:

- Need to ask for memory and return it, more explicitly
- pros:
    - more efficient
    - better in real–time applications
- cons:
    - more work for the programmer
    - more prone to errors

Garbage collection:

- Memory freed automatically
- pros:
    - less work for the programmer
    - more difficult to make mistakes
- cons:
    - less efficient
    - can lead to sudden slowdowns

Memory management: manual vs. garbage collection

Manual:

- Need to ask for memory and return it, more explicitly
- pros:
    - more efficient
    - better in real–time applications
- cons:
    - more work for the programmer
    - more prone to errors

Garbage collection:

- Memory freed automatically
- pros:
    - less work for the programmer
    - more difficult to make mistakes
- cons:
    - less efficient
    - can lead to sudden slowdowns

Rust has many features to avoid memory management errors

Rust

- high–level
- compiled
- static type checking
- manual memory management

Most important difference between Python and Rust?

Most important difference between Python and Rust?

Braces, { }, for code formatting!

Most important difference between Python and Rust?

Braces, { }, for code formatting!

- How do we denote blocks of code?
  - Python: indentation
  - Rust: `{...}`

```rust
fn hi() {
    println!("Hello!");
    println!("How are you?");
}
```

Most important difference between Python and Rust?

Braces, { }, for code formatting!

- How do we denote blocks of code?
  - Python: indentation
  - Rust: `{...}`

```rust
fn hi() {
    println!("Hello!");
    println!("How are you?");
}
```

- Don't be afraid of braces!!! You'll encounter them in C, C++, Java, Javascript, PHP, Rust, ...

Discussion section today

Git / Github demo