タイトル（仮）
RustとPythonのメモリ管理手法の比較を通した，Rust→Python プログラム変換手法の検討

住井・松田研究室　三上 陽向

内容（仮）

1. 研究の動機
2. 方法
3. 例
4. 結論･考察
5. 今後の展望

1. 研究の動機

- Python : 動的メモリ管理・制約がやや甘い
- Rust : 静的メモリ管理・非常に厳しい制約

Python プログラムでは，プログラマの不注意によってはメモリリークが発生しうる．組み込み機器などの限られたメモリ環境でのメモリリークの発生は，バグの原因となる．

Rust の静的検査をクリアしたプログラムを，Rust のメモリ管理手法を維持した状態でPythonに変換することができれば，Pythonしか使用できないような開発環境でもメモリ安全なプログラムを生成することができるのではないか．

2. 方法

最終的には機械的にコード変換を行うことを想定し，Rustの様々なメモリ管理手法を代表するいくつかのコード例とそのメモリ管理手法を模倣したPythonコード例を比較し，コード変換の手法および問題点等について検討する.

例3.1 所有権の移動


```rust
fn main() {

    let s1 : String = String::from("s1");

    let s11 = s1;
    println!("s11 : {}", s11);

}
```

```python
def main():

    s1 = "s1"

    s11 = s1
    del s1

    print(f"s11 : {s11}")
```

let s11 = s1; で所有権が移動，s1はそれ以降使用不可
main関数(s1のスコープ)を抜けるとs1のデータを解放.

Pythonではオブジェクトを複製.
Rustではオブジェクトは複製せず，所有権が移動するだけ.
明示的に del s1 をすることで名前を使用不可にし，s1が
格納されているメモリはRC = 0となることにより解放される.
これによりRustの所有権移動を模倣.

例3.1 所有権の移動ー解放のタイミングの確認


```rust
fn main() {
    // 解放されるタイミングの確認用
    check_drop();
    println!("Check if s2 is dropped before this comment.");
}

// 所有権を持つオブジェクトのメモリがいつ解放されるかは,
// 構造体とそれに対するDropメソッドを定義することで確認できる.
struct MyString(String);

impl Drop for MyString {
    fn drop(&mut self) {
        println!("Dropping MyString: {}", self.0);
    }
}

fn check_drop() {

    let s2 = MyString(String::from("s2"));

    let s21 = s2;                    // 所有権移動 s2は以後無効
    println!("s21 : {}", s21.0);

    println!("Check if s2 is dropped after this comment.");
}                                    // LT終了によるメモリ解放 : s21
```

```
s21 : s2
Check if s2 is dropped after this comment.
Dropping MyString: s2
Check if s2 is dropped before this comment.
```

例3.2 借用


```rust
1   fn main() {
2       let s2 : &str = "s2";
3
4       let s21 = s2;
5       println!("{}", s21);
6       println!("{}", s2);
7   }
```

```
s2
s2
```

借用型に所有権は存在しないので，4行目以降もs2は使用可能.

“s2”は静的領域に保存されており，ライフタイムはプログラムが終了するまで（’static）である.

```python
1   def main():
2       s2 = "s2"
3
4       s21 = s2
5       print(f"{s21}")
6       print(f"{s2}")
```

```
s2
s2
```

Pythonでは，参照型は存在しないので，s2のコピーs21を作成する．この場合，s2, s21 のオブジェクトidは同じであり，s2, s21は同じメモリ領域を指している．”s2”オブジェクトが解放されるのはプログラム終了時であり，Rustと挙動の違いはない.

例3.3.1 可変借用


```rust
fn main() {
    let mut s1 = vec![String::from("s1")];

    let s2 = &mut s1;
    s2[0].push_str(" : modified");
    println!("{}", s2[0]);
    println!("{}", s1[0]);
}
```

```
s1 : modified
s1 : modified
```

Rustにおけるミュータブルなオブジェクトであるベクタ(≒リスト)s1 とその可変参照としてs2を作る．ここで，s2[0] に &str をプッシュすると，s2[0] および元の s1[0] も変更されている．

```python
def main():
    s1 = ["s1"]

    s2 = s1
    s2[0] += " : modified"
    print(s2[0])
    print(s1[0])
```

```
s1 : modified
s1 : modified
```

Pythonでも，s2[0] にStringをプッシュすると，s2[0] だけでなく元の s1[0] も変更されている．

例3.3.2 可変借用（変換できない例）


```rust
fn main() {
    let mut s1 = String::from("s1");

    let s2 = &mut s1;
    s2.push_str(" : modified");
    println!("{}", s2);
    println!("{}", s1);
}
```

```
s1 : modified
s1 : modified
```

String は Rust ではミュータブルである.

```python
def main():
    s1 = "s1"

    s2 = s1
    s2 += " : modified"
    print(f"{s2}")
    print(f"{s1}")
```

```
s1 : modified
s1
```

String は Python ではイミュータブルであり，s2 に String をプッシュした時点で別のオブジェクトになってしまう.

Rustではミュータブルで，Pythonではイミュータブルであるような型（数，文字列，タプル）では，同様の問題が発生する． 変換を実現するにはPython上でミュータブルな数・文字列・タプルを作るか，それに準ずる機能を実現しなければならない

例3.4.1 クローン


```rust
fn main() {

    let s1 = String::from("s1");
    let mut s2 = s1.clone();
    s2.push_str(" : modified");
    println!("{}", s1);
    println!("{}", s2);

}
```

```
s1
s1 : modified
```

s1のクローンオブジェクトs2を作成し，s2 に &str をプッシュすると，s1 は変更されず，s2 のみが変更されている.

```python
def main():

    s1 = "s1"
    s2 = s1
    s2 += " : modified"
    print(f"{s1}")
    print(f"{s2}")
```

```
s1
s1 : modified
```

Python は s2 に String をプッシュした時点で別オブジェクトが生成されるため，オブジェクトが複製されるタイミングは1行分異なるが，ほぼ等しい挙動・メモリ管理をしているといえる.

例3.4.2 クローン


```rust
fn main() {
    let s1 = vec![String::from("s1")];

    let mut s2 = s1.clone();
    s2[0].push_str(" : modified");
    println!("{}", s1[0]);
    println!("{}", s2[0]);

}
```

```
s1
s1 : modified
```

ベクタは Rust ではミュータブルであるが，s2 はクローンされたオブジェクトなので s1 に影響を与えない．

```python
import copy

def main():
    s1 = ["s1"]

    s2 = copy.deepcopy(s1)
    s2[0] += " : modified"
    print(s1[0])
    print(s2[0])
```

```
s1
s1 : modified
```

Pythonでは，s2 = s1 としても同じオブジェクトへの参照をコピーするだけで，元のオブジェクトがミュータブルだと s2[0] の変更が s1[0] にも影響を与えてしまう．
copyモジュールのdeepcopyを実行することで，コンテナオブジェクトとその各要素を，新オブジェクトとして再帰的にコピーできる．

例3.5.1 定数，グローバル変数


```rust
static HELLO: &str = "Hello";
const WORLD: &str = "World";
static mut COUNTER: i32 = 0;

fn incr_count() {
    unsafe {
        COUNTER += 1;
    }
}

fn main() {
    println!("{}, {}!", HELLO, WORLD);

    // 可変な静的変数を使用する際はunsafeブロックが必要
    unsafe {
        incr_count();
        println!("Count: {}", COUNTER);
    }
}
```

```
Hello, World!
Count: 1
```

```python
HELLO = "Hello"
WORLD = "World"
COUNTER = 0

def incr_count():
    global COUNTER
    COUNTER += 1

def main():
    print(f"{HELLO}, {WORLD}!")

    incr_count()
    global COUNTER
    print(f"Count: {COUNTER}")
```

```
Hello, World!
Count: 1
```

Pythonでは，グローバル変数を使用する際は宣言する必要がある．また，イミュータブルな値の，破壊的代入による値の変更を禁止する記述は簡単には書けない．

例3.5.2 Pythonにおけるクラスを利用した値の変更の抑制


```python
class ImmutableObject:
    def __init__(self, **kwargs):
        for key, value in kwargs.items():
            # 親クラスの__setattr__を使用することでErrorを起こさない
            # デフォルトの__setattr__メソッドを使用
            # self.name = value のようなことをしている
            # 属性attrがself.__dict__に追加され、値valueが設定される
            super().__setattr__(key, value)

    # obj.attr = value のたびに自動的に呼ばれる関数
    def __setattr__(self, name, value):
        raise AttributeError("Cannot modify immutable object")

    def __delattr__(self, name):
        raise AttributeError("Cannot delete attribute from immutable object")


if __name__ == "__main__":
    obj = ImmutableObject(x=10, y=20)
    print(obj.x)  # 10
    # obj.x = 30  # AttributeError: Cannot modify immutable object
```

例3.6.1 スコープ


```rust
fn main() {
    let s1 = String::from("s1");

    {
        let s11 = s1;
        println!("{}", s11);
    }
    // println!("{}", s1;  Error!
}
```

```python
def scope(s1):
    s11 = s1
    print(s11)

def main() :
    s1 = "s1"

    scope(s1)
    del s1
```

Rust ではスコープを波括弧で記述できる.

s11 はデータ String::from(“s1”) の所有権を5行目で受け取るため，s1は使用不可となる.

7行目でs11はスコープを抜けるため使用不可となり，これにより String::from(“s1”) のデータは解放される.

Python では関数を用いてスコープを再現する方法があるが，s1 の解放のタイミングがRustと比べて遅れてしまっている．今回は s11 は s1 のコピーであるため，新たなオブジェクトを生成するわけではない．しかし，例えばこの後スコープ内でs11の値を変更すると，s1 と s11 はそれぞれ別のデータを保持することになるため，Rust より多くのメモリ領域を必要とすることになる．これはメモリ管理を模倣するという点では大きな問題である.

例3.6.2 スコープ


```rust
fn main() {
    let s1 = String::from("s1");

    {
        let s11 = s1;
        println!("{}", s11);
    }
    // println!("{}", s1;  Error!
}
```

```python
from contextlib import nullcontext

def main():
    s1 = "s1"

    with nullcontext():
        s11 = s1
        del s1
        print(s11)
        del s11  # del しないと11行目以降も使用可能

if __name__ == "__main__":
    main()
```

https://docs.python.org/ja/3/library/contextlib.html

nullcontext を使用したコンテキストマネージャでもスコープを作ることができる. これを使用したスコープの模倣のほうがより Rust に近いメモリ管理を実現できる.

例3.7 ライフタイム


```rust
fn main() {
    let outer_ref;
    {
        let inner_value = String::from("inner value");
        outer_ref = &inner_value;
    }
}
```

```python
from contextlib import nullcontext

def main():

    outer_ref = None

    with nullcontext():
        inner_value = "inner value"
        outer_ref = inner_value
        del outer_ref, inner_value
```

outer_ref は main関数のスコープ内をライフタイムとするように定義されているが，5行目の，内側の変数への参照により，outer_ref のライフタイムは4 ~ 6 行目のスコープに圧縮されている.

Python では，データはライフタイムではなく参照カウントにより管理されており，長生き / 短命 という概念はない．そのため，Rust プログラムではいつ解放されるかを判断し，適切な位置に del を挿入することが必要である.

例3.8.1 Rc


```rust
use std::rc::Rc;

fn main() {

    let a = Rc::new(1);
    let b = Rc::clone(&a);
    println!("a: {}, b: {}", a, b);
    println!("Reference count: {}", Rc::strong_count(&a));

}
```

```python
import sys

def main():
    a = 1
    b = a
    print(f"a = {a}, b = {b}")
    print(f"Reference count: {sys.getrefcount(a)}")

    # https://docs.python.org/ja/3/library/sys.html
    # Immortal objects have very large refcounts
    # that do not match the actual number of references
    # to the object.
```

例3.8.2 Rc とスコープ


```rust
use std::rc::Rc;

struct MyStruct {
    value: i32,
}

impl Drop for MyStruct {
    fn drop(&mut self) {
        println!("Dropping MyStruct with value: {}", self.value);
    }
}

fn main() {
    {
        let a = Rc::new(MyStruct { value: 1 });
        let b = Rc::clone(&a);
        println!("a: {}, b: {}", a.value, b.value);
        println!("Reference count: {}", Rc::strong_count(&a));
    }

    println!("Exited scope");
}
```

```
a: 1, b: 1
Reference count: 2
Dropping MyStruct with value: 1
Exited scope
```

```python
import sys
from contextlib import nullcontext

class MyStruct:
    def __init__(self, value):
        self.value = value

    def __del__(self):
        print(f"Dropping MyStruct with value: {self.value}")

def main():

    with nullcontext():
        a = MyStruct(1)
        b = a
        print(f"a: {a.value} b: {b.value}")
        print(f"Reference count: {sys.getrefcount(a)-1}")
        del a, b

    print("Exited scope")
```

```
a: 1 b: 1
Reference count: 2
Dropping MyStruct with value: 1
Exited scope
```

例3.9 Rc と循環参照

例3.10.1 Arc

例3.10.2 Arc と Mutex

4. 結論･考察

5. 今後の展望