プログラミング1

(第9回) 命名規則、KISS原則＆DRY原則、抽象化

1. プログラミング補足
    1. 変数名・ファイル名の命名規則
    2. KISS原則、DRY原則
2. Chapter 4.3 Recursion
    1. 実際の動作とスタックフレームの対応
    2. 普通の反復処理 vs. 再帰呼び出し
    3. 迷路探索の例
3. Chapter 4.4 Global Variables
4. まとめ
5. デモ？（時間あれば）

講義ページ: http://ie.u-ryukyu.ac.jp/~tnal/2019/prog1/

変数名・ファイル名の命名規則

- 変数名やファイル名に使える文字
  - 原則として「英数字」と「_ (underscore)」。
  - 大文字小文字は区別される。
  - 冒頭に数字は使えない。
- 変数名・ファイル名を適切に選択する
  - 「適切」とは？
    - **Level 1: その変数が表す用語の英単語（小文字）を使う。**
    - **Level 2: 複数単語で命名したいなら「_ (underscore)」で繋げて書く。**
    - Level 3: 同じモジュール・クラス内では統一規約を採用する。
    - Level 4: 英単語の微妙なニュアンスの差に気をつける。
  - 規約の例: Google Python Style Guide
    - https://google.github.io/styleguide/pyguide.html
    - Naming
      - Modules（ファイル名）: lower_with_under
      - Local Variables（変数名）: lower_with_under

KISS原則とDRY原則

- KISS原則
  - Keep it simple, stupid!
  - 小さく作り、組み合わせる。
    - 一つの関数は一つの作業をこなす。
  - 各部品(関数)をテストする。
    - 検証・再現性を意識する。

目安:
・1関数=数十行程度
・50行ぐらいになったら分割できないか考えてみよう。
・長過ぎるブロックや関数は読みづらく、バグに気づきにくく、再利用しにくい。

- DRY原則
  - Don't repeat yourself.
  - 繰り返しを避ける。

目安:
・同じ行or類似行が数回出てきたら、関数化することを検討しよう。
・同一機能を何度もコードやブロックとして繰り返して書くと、同一バグがあると全ての関連箇所を修正する必要があるし、機能改善をする際にも同様の手間がかかる。

プログラミング1

(第9回) 命名規則、KISS原則＆DRY原則、抽象化

1. プログラミング補足
    1. 変数名・ファイル名の命名規則
    2. KISS原則、DRY原則
2. Chapter 4.3 Recursion
    1. 実際の動作とスタックフレームの対応
    2. 普通の反復処理 vs. 再帰呼び出し
    3. 迷路探索の例
3. Chapter 4.4 Global Variables
4. まとめ
5. デモ？（時間あれば）

命名規則＆2大原則を意識して、読みやすいコードとなるよう工夫してみよう。

Chapter 4.3 の補足

4.3 Recursion (再帰)

4.3 Recursion (再帰)
factorial function (階乗関数)

```
# コード例1
# これまでの反復を利用
def factI(n):
    """iterative function
    >>> factI(1)
    1
    >>> factI(3)
    6
    """
    result = 1
    while n > 1:
        result = result * n
        n -= 1

    return result
```

```
# コード例2
# 自分自身を再帰的に呼び出す
def factR(n):
    """recursive function
    >>> factR(1)
    1
    >>> factR(3)
    6
    """
    if n == 1:
        return n
    else:
        return n * factR(n-1)
```

factR()という関数定義の中で、
factR()自身を呼び出している。

実際の動作とスタックフレームの対応

```
# コード例2-2
# 自分自身を再帰的に呼び出す
def factR(n):
    """recursive function
    >>> factR(1)
    1
    >>> factR(3)
    6
    """
    if n == 1:
        return n
    else:
        return n * factR(n-1)

# 実行
result = factR(3)
```

- トップレベル (Stack no.1)
  - factR()
  - result
- factR(3)
  - 1回目の呼び出し (Stack no.2)
  - n = 3
  - factR(2)　# factR(3)は継続中
- factR(2)
  - 2回目の呼び出し (Stack no.3)
  - n = 2
  - factR(1)　# factR(2)は継続中
- factR(1)
  - 3回目の呼び出し (Stack no.4)
  - n = 1
  - return 1　# factR(1)が終了

普通の反復処理 vs. 再帰呼び出し

- 再帰のメリット
  - 同じ構造に対して手続きを書くなら、シンプルなコードになることがある。
    - シンプル＝読みやすい、書きやすい、バグに気づきやすい
    - 「同じ構造」の例
      - 木構造、グラフ、、、
- 再帰のデメリット
  - Stack Overflow (高コストになりがち)
    - 関数が終わるまでスタックフレームが積み重なる（廃棄されず、メモリに残り続ける）。

迷路探索の例（概要）

- 再帰
  - 「現在地点から時計回りに確認。進めるなら先に進む。」（深さ優先探索）
- コード
  - https://github.com/naltoma/python_demo_module
- 動かし方
  - case 1

    % python maze_simple.py
  - case 2

    % python

    >>> import maze_simple

    >>> maze_simple.test_play()

Case1のファイル実行の場合、
if __name__ == '__main__':
のmainブロックが実行される。

Case2のimportをした場合、
if __name__ == '__main__':
のmainブロックは実行されない。

import時にも自動実行して欲しいなら、これまで通りの書き方でOK。自動実行して欲しくないなら、mainブロックを使おう。変数「__name__」は自動設定される特別な変数の一つ。中身は各自で確認してみよう。

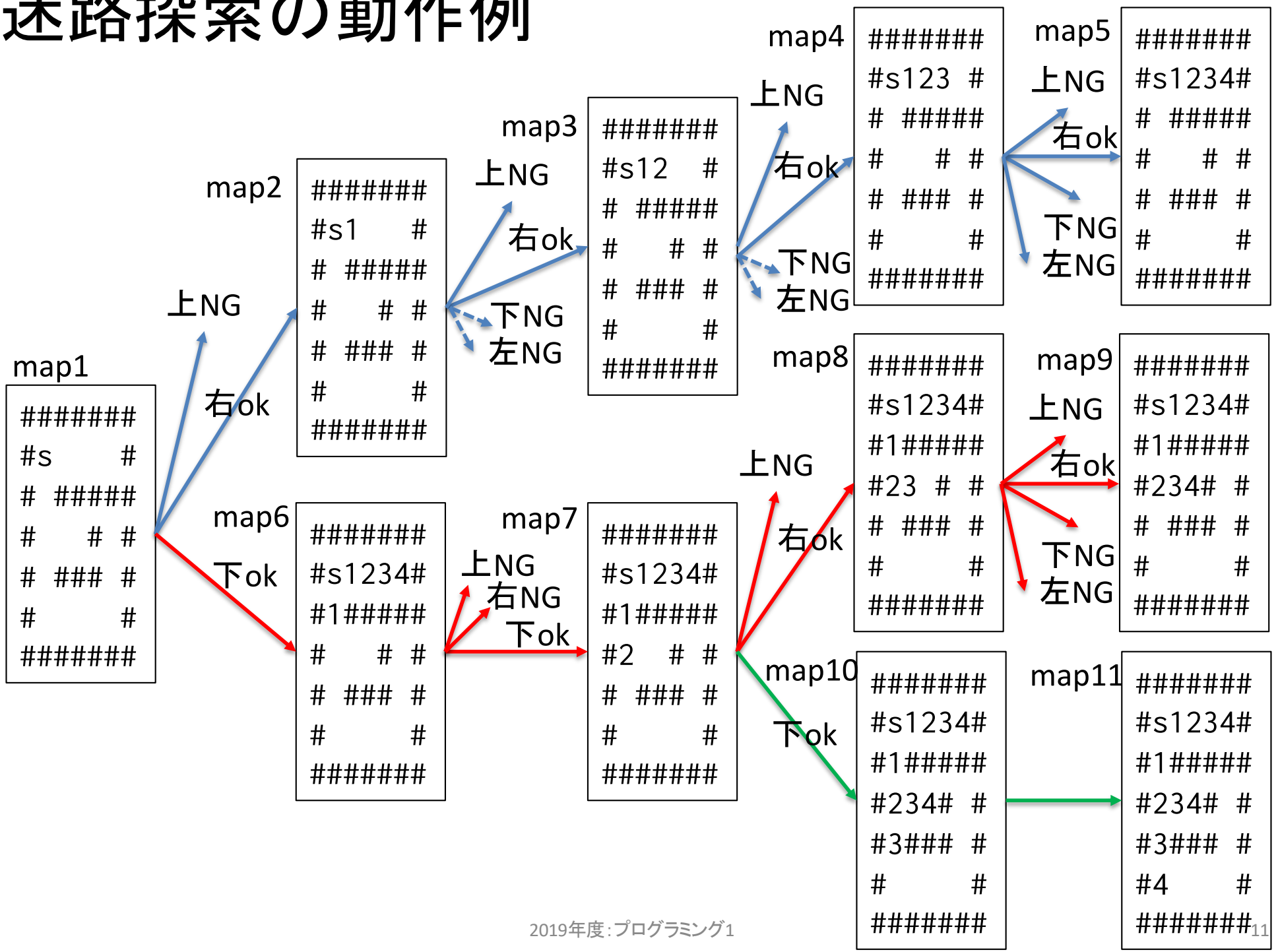
迷路探索の例: コードの再帰部分(抜粋)

```
def walk_map_with_step_num(map, y, x, step_num):

    if is_upward(map, y, x) == True:
        step_num += 1
        map[y-1][x] = step_num
        walk_map_with_step_num(map, y-1, x, step_num)

    if is_rightword(map, y, x) == True:
        step_num += 1
        map[y][x+1] = step_num
        walk_map_with_step_num(map, y, x+1, step_num)

    if is_downward(map, y, x) == True:
        step_num += 1
        map[y+1][x] = step_num
        walk_map_with_step_num(map, y+1, x, step_num)

    if is_leftward(map, y, x) == True:
        step_num += 1
        map[y][x-1] = step_num
        walk_map_with_step_num(map, y, x-1, step_num)
```

現在地map[y][x]の上方向が未記入なら、歩数を1つ増やし、mapに歩数を記入する。

記入し終えたら、新しい場所に移動して、同じ操作を繰り返す。

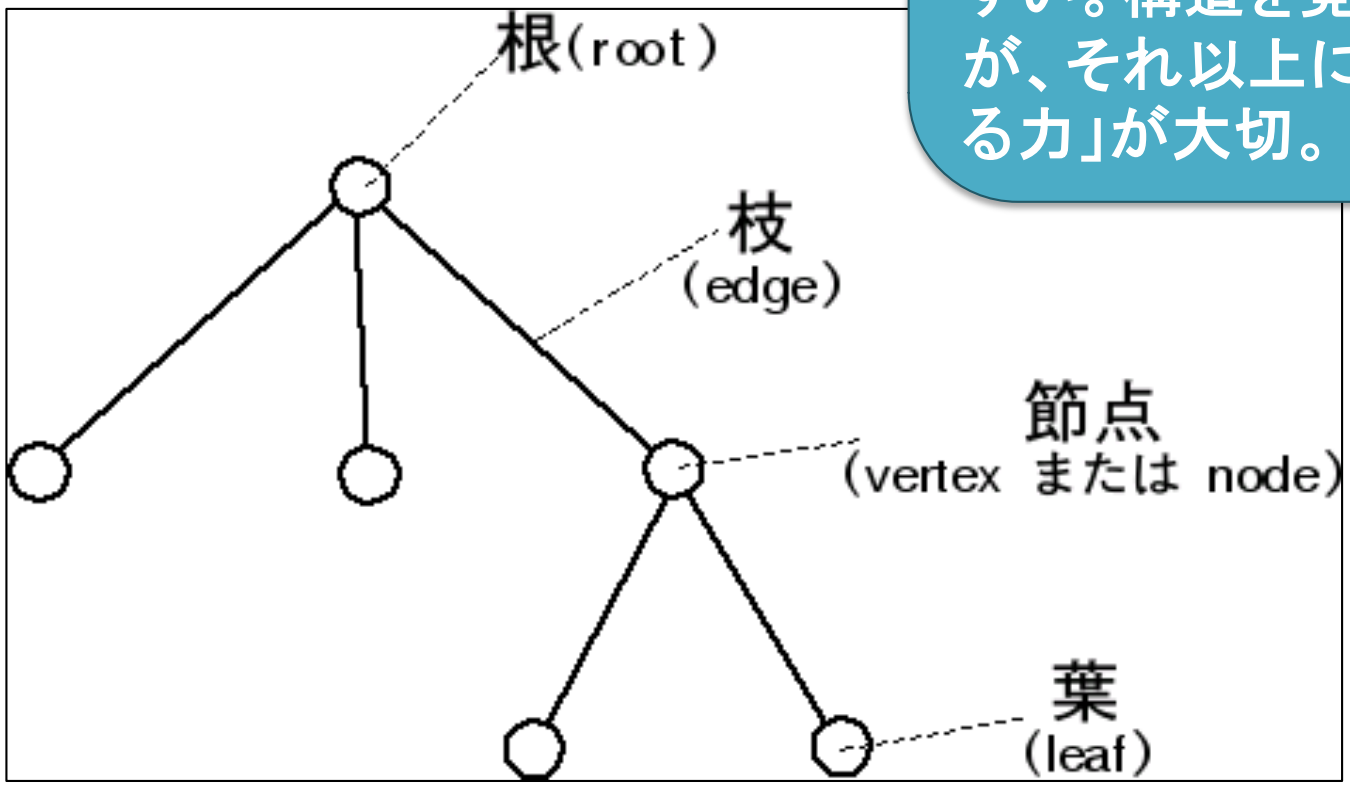
迷路探索の動作例


map1
#######
#s #
#####
# #
#
# #
#######
上NG
右ok
map2
#######
#s1 #
#####
# #
#
# #
#######
上NG
右ok
下NG
左NG
map3
#######
#s12 #
#####
# #
#
# #
#######
上NG
右ok
下NG
左NG
map4
#######
#s123 #
#####
# #
#
# #
#######
上NG
右ok
下NG
左NG
map5
#######
#s1234#
#####
# #
#
# #
#######
下ok
map6
#######
#s1234#
#1#####
# #
#
# #
#######
上NG
右NG
下ok
map7
#######
#s1234#
#1#####
#2 # #
#
# #
#######
上NG
右ok
下ok
map8
#######
#s1234#
#1#####
#23 # #
#
# #
#######
上NG
右ok
下NG
左NG
map9
#######
#s1234#
#1#####
#234# #
#
# #
#######
下ok
map10
#######
#s1234#
#1#####
#234# #
#3### #
# #
#######
map11
#######
#s1234#
#1#####
#234# #
#3### #
#4 #
#######

木構造

閉路がなければ木構造。
閉路があるとグラフ。

見た目そのものではなく、構造を捉える（設計できる）と実装しやすい。構造を覚えることも重要だが、それ以上に「抽象化して考える力」が大切。



出典: https://ja.wikipedia.org/wiki/木_(数学)

プログラミング1

(第9回) 命名規則、KISS原則＆DRY原則、抽象化

1. プログラミング補足
    1. 変数名・ファイル名の命名規則
    2. KISS原則、DRY原則
2. Chapter 4.3 Recursion
    1. 実際の動作とスタックフレームの対応
    2. 普通の反復処理 vs. 再帰呼び出し
    3. 迷路探索の例
3. Chapter 4.4 Global Variables
4. まとめ
5. デモ？（時間あれば）

命名規則＆2大原則を意識して、**読みやすいコード**となるよう工夫してみよう。

対象を抽象的に捉えることで、実装しやすくなる。まずは**再帰関数の動作を理解できるようになろう**。

Chapter 4.4 の補足

4.4 Global Variables (大域変数)

原則禁止！！

4.4 Global Variables (大域変数)

- 大域変数として使いたい変数を「global」宣言してから使う
  - コード例: テキスト参照
- どこからでも参照できる変数
  - 参照できるからといって使いまくると、「この変数がどこからアクセスされるのか」を読み解くことが困難になる。
  - 困難＝理解し難い、バグの温床になりがち。
- 原則
  - 使わないに越したことはない。
  - 授業としては「原則禁止」。どうしても使いたいなら、その理由を解説した上で使用すること。
  - 使うとしても、最小限に抑える。

プログラミング1

(第9回) 命名規則、KISS原則＆DRY原則、抽象化

1. プログラミング補足
    1. 変数名・ファイル名の命名規則
    2. KISS原則、DRY原則
2. Chapter 4.3 Recursion
    1. 実際の動作とスタックフレームの対応
    2. 普通の反復処理 vs. 再帰呼び出し
    3. 迷路探索の例
3. Chapter 4.4 Global Variables
4. まとめ
5. デモ？（時間あれば）

命名規則＆2大原則を意識して、読みやすいコードとなるよう工夫してみよう。

対象を抽象的に捉えることで、実装しやすくなる。まずは再帰関数の動作を理解できるようになろう。

通常は大域変数を使う必要はありません。使わないで下さい。

まとめ

- KISS原則とDRY原則
  - 1関数はシンプル（単機能）にし、機能をテストする。orテスト駆動開発。
  - 繰り返してるコードがあれば、繰り返さずに書けないか考えてみる。
- 再帰
  - 関数自身を繰り返し呼ぶことで処理しやすいケースがある。
- 木構造・グラフ構造
  - 「基本構造」はツール。その構造に落とし込む形でモデル化できると、想定漏れを防ぎやすい（コードに落としやすい）。
- 大域変数
  - 原則禁止

モデル化って何だろう？
＊学習教育目標にあり。

デモ？（パースしてみる）

```
#name,HP
"slime":5
"goblin":10
```

- 読み込んだ行の冒頭が#ならば、処理対象外とする。
- 読み込んだ行の冒頭が"ならば、「"敵の名前":数字」という書式で、名前とHPが記述されているとする。
- 「"slime":5」を {'slime':5} としてdict型オブジェクトとして保存したい。