先週の補足2件

- ユニットテスト（doctest）の利用
  - テストの目的
  - 書き方
  - 実行方法
- モジュールの利用
  - モジュールのメリット
  - モジュール＝「*.py」
  - importとfromを用いたモジュール読み込み

先週の補足2件

- ユニットテスト（doctest）の利用
  - テストの目的
  - 書き方
  - 実行方法
- モジュールの利用
  - モジュールのメリット
  - モジュール＝「*.py」
  - importとfromを用いたモジュール読み込み

doctestの書き方

- ユニットテストの書き方
  - docstring形式ドキュメント中に、インタプリタの出力を含めて実行例と結果例を記入する。
  - インデントを揃えること。
  - スペースの有無に注意。
- テストしたい関数の例

```
def add(x, y):
    result = x + y
    return result
```

- テストを含めた例

```
def add(x, y):
    """
    >>> add(1, 2)
    3
    """
    result = x + y
    return result

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

テストの実装

ファイル実行時にユニットテストを実行するという指示。

doctestの実行方法（2通り）

- 通常実行
  - 「python3 ファイル名」としてファイル実行するだけでテストを実行する。
  - テスト失敗があるときだけ、テスト結果を出力する。（全てのテストが成功する場合、テスト結果は出力しない）

- 詳細出力実行
  - 「python3 ファイル名 -v」のように、-vオプション付きで実行する。
  - テストの成功失敗を問わず、テスト結果詳細を出力する。

何のためにユニットテストを書くのか？

- 関数が想定通りに動くことを「検証しやすくする」ため。
  - 毎回手動確認するのは辛い。
  - 「想定通り」の質が確認者に強く依存する。
  - 「動作例」を記述することで、想定している使い方を示すことにもなる。
- リファクタリング
  - 機能を保ったまま、コードを修正。

- テスト駆動開発
  - 背景: 開発現場でよくある状況
    - 「こういう入力が与えられた時に、こういう出力をする関数を書いて」
    - 入力と出力は分かっている。or 分からないなら、コードを書き始める前に明確にする。
  - テストを先に書くことで、その関数をどう動作させたいかを明確にする。
  - 書いたテストが通るようにコードを書く。

参考:
エンジニアのスキルを伸ばす「テスト駆動開発」を学んでみよう:
https://thinkit.co.jp/story/2014/07/30/5097

ユニットテスト演習

- https://github.com/naltoma/python_intro/blob/master/tutorial-doctest.md

先週の補足2件

- ユニットテスト（doctest）の利用
  - テストの目的
  - 書き方
  - 実行方法
- モジュールの利用
  - モジュールのメリット
  - モジュール＝「*.py」
  - importとfromを用いたモジュール読み込み

4.5 Modules (モジュール, 部品)

- コードを保存したファイル = モジュール
- 使用例1

```
% python3
>>> import week6_doctest
>>> week6_doctest.add(1, 2)
3
```

自作モジュールをimportで読み込む

「モジュール名.**」でモジュール内の部品を利用。

- 使用例2

```
% python3
>>> import week6_doctest as wd
>>> wd.add(1, 2)
3
```

モジュール名に別称を付ける

- 使用例3

```
% python3
>>> from week6_doctest import *
>>> add(1, 2)
3
```

モジュール内の部品を指定して読み込む。「*」は全ての部品。

- 読み込んだモジュールはhelp()で簡易ドキュメントを参照できる。

```
>>> import week6_doctest as wd
>>> help(wd)
```

pydoc3で読めるドキュメントと、内容は同じ。

何のために「モジュール」があるのか？

- 第三者が作成したモジュールを、再利用しやすくするため。
  - 関数だけだと、同一ファイル内でしか再利用できない。
  - モジュールなら、別ファイルでも再利用できる。
- なるべくファイル編集させないようにするため。
  - 人は過ちを侵してしまう。
  - 何気ない編集のつもりが、バグに繋がることも。
  - モジュール（別ファイル）として利用するなら、少なくともファイル編集に伴うバグは発生しない。

モジュールを利用する例

- https://github.com/naltoma/python_demo_module
- ユニットテスト演習で使った例題
  - my_math.py #モジュールとして用意したコード
  - import_case1.py #import例
  - import_case2.py #from+import例

先週の補足2件

- ユニットテスト（doctest）の利用
  - テストの目的
  - 書き方
  - 実行方法

ユニットテストを用いた開発に慣れよう

- モジュールの利用
  - モジュールのメリット
  - モジュール＝「*.py」
  - importとfromを用いたモジュール読み込み

from, importによるモジュール読み込みを使えるようになろう。

ここから新しい内容

プログラミング1

(第7回) KISS原則とDRY原則、再帰と木構造

1. 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. 演習
6. 宿題

講義ページ: http://ie.u-ryukyu.ac.jp/~tnal/2017/prog1/

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10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7回) KISS原則とDRY原則、再帰と木構造

1. 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. 演習
6. 宿題

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。

講義ページ: http://ie.u-ryukyu.ac.jp/~tnal/2017/prog1/

Chapter 4.3 の補足

4.3 Recursion (再帰)

4.3 Recursion (再帰)
p.45, 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

    ```
    % python3 maze_simple.py
    ```
  - case 2

    ```
    % python3
    >>> import maze_simple
    >>> maze_simple.test_play()
    ```

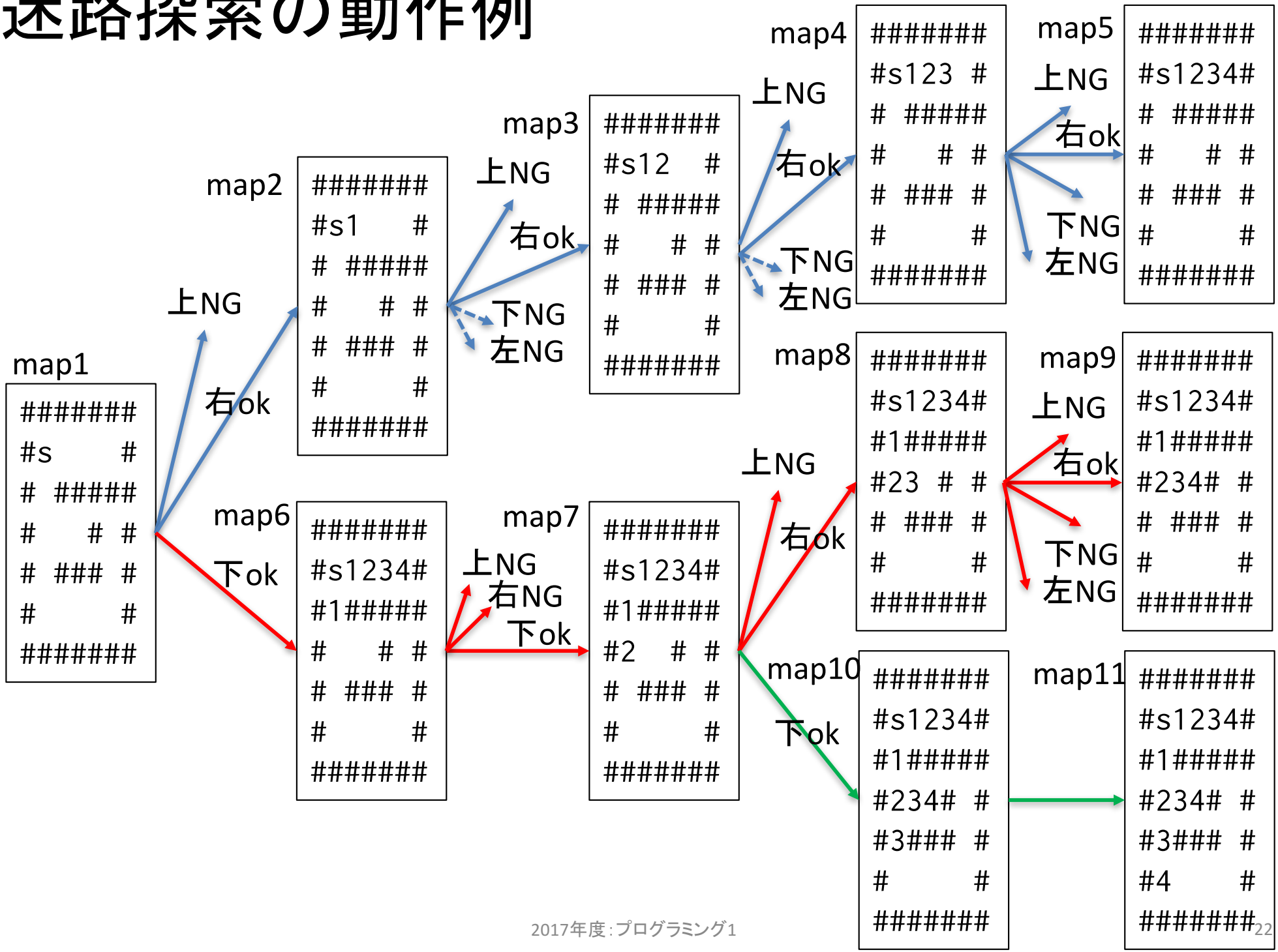
迷路探索の例: コードの再帰部分(抜粋)

```
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。

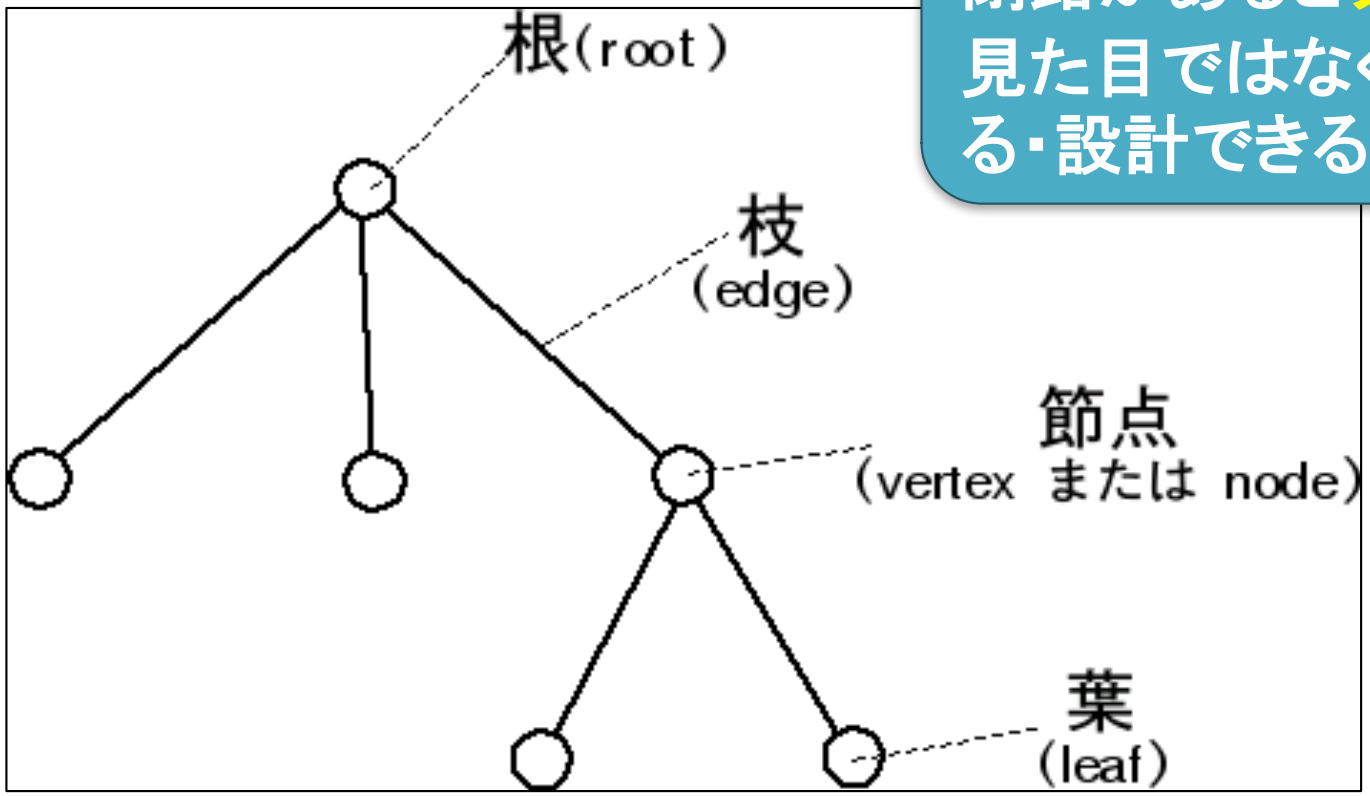
記入し終えたら、新しい場所に移動して、同じ操作を繰り返す。

迷路探索の動作例


map1
#######
#s #
#####
# # #
#
# #
#######
上NG
右ok
map2
#######
#s1 #
#####
# # #
#
# #
#######
下ok
map6
#######
#s1234#
#1#####
# # #
#
# #
#######
上NG
右ok
下NG
左NG
map3
#######
#s12 #
#####
# # #
#
# #
#######
上NG
右ok
下NG
左NG
map7
#######
#s1234#
#1#####
#2 # #
#
# #
#######
上NG
右NG
下ok
map4
#######
#s123 #
#####
# # #
#
# #
#######
上NG
右ok
下NG
左NG
map5
#######
#s1234#
#####
# # #
#
# #
#######
上NG
右ok
下NG
左NG
map8
#######
#s1234#
#1#####
#23 # #
#
# #
#######
上NG
右ok
map9
#######
#s1234#
#1#####
#234# #
#
# #
#######
上NG
右ok
下NG
左NG
map10
#######
#s1234#
#1#####
#234# #
#3### #
# #
#######
下ok
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。
見た目ではなく、構造を捉える・設計できると実装しやすい。



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

プログラミング1

(第7回) KISS原則とDRY原則、再帰と木構造

1. 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. 演習
6. 宿題

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。**

講義ページ: http://ie.u-ryukyu.ac.jp/~tnal/2017/prog1/

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7回) KISS原則とDRY原則、再帰と木構造

1. 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. 演習
6. 宿題

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。

講義ページ: http://ie.u-ryukyu.ac.jp/~tnal/2017/prog1/

プログラミング1

(第7回) KISS原則とDRY原則、再帰と木構造

1. 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. 演習
6. 宿題

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。

講義ページ: http://ie.u-ryukyu.ac.jp/~tnal/2017/prog1/

まとめ

- ユニットテスト
  - 動作検証の自動化、テスト駆動開発、リファクタリング。
- 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。

ペアプロ演習

- 演習1～演習4: 初めてのペア・プログラミング
  - この演習は終了。回答例を「google drive/ex1/0-example」に掲載しました。
  - 自分たちの回答と見比べてみよう。回答例が理解できない場合には質問しよう。
- 演習5: 数当てゲーム1 (大小ヒント付き) を実装してみよう。
  - Google drive内の「**ex5_2**」。
  - 今日から新しいペアになるので、演習結果を記録するページを新しく用意しました。

ペアプロ補足

- ペアプログラミングのやり方
  - 簡易版4ステップ
    - 1. 作業を決める
    - 2. 役割を決める
      - driver: 仕事を終わらせることに専念
      - observer: 横から観察し、疑問・改善・簡潔化など大局的な問題について考える
    - 3. タスクに取り組む
      - 頻繁に同期をとり、お互い何をやっているか把握する
        - » 相手が何やってるか分からなくなったら尋ねよう。
      - 「一人で悩む」のは十秒程度に留める
      - 一緒に相談しながら考える練習
    - 4. 交代する

分業ではない（observer=観察しながら気づいたことをコメント、分からないことを質問）

二人で2,3分考えても分からない場合には、手を上げて質問しよう

宿題

- 復習: 適宜(これまでの内容)
- 予習: 教科書読み
  - 5章
    - 5.1 Tuples
    - (5.2 Lists and Mutabilities)　＊余裕があれば。
- 復習・予習(オススメ): paiza, progate
  - プログラミングスキルチェック　＊レベル設定のある課題集
    - https://paiza.jp/challenges/info

参考文献

- 教科書: Introduction to Computation and Programming Using Python, Revised And Expanded Edition
- Python 3.5.1 documentation, https://docs.python.org/3.5/index.html
- doctest — Test interactive Python examples, https://docs.python.org/3/library/doctest.html
- エンジニアのスキルを伸ばす「テスト駆動開発」を学んでみよう, https://thinkit.co.jp/story/2014/07/30/5097