Pure Code Essentials
モデル化による仮説形成と、コードによる問題解決

奥村　稔
OKUMURA Minoru

2017/01/12 SUGOROKU version 1.0

目次

1 概要

コンピュータにおけるプログラミングは、問題を一つの仮説としてモデル化し、それをシミュレートすることで[*1]結果を確認し、ヒューリスティック[*2]に解決しようとする行為です。さらにまた、コーディングしてモデル化することは言語活動であり、[*3]ヒューリスティックな学習活動はアクティブラーニング[*4]でもあります。プロトタイプ[*5]から始め、次第に改良を重ねることで問題解決に近づいていこうとするプログラミングを通して、今求められているそうした学習スタイルを取り込んでみましょう。

どのようなプログラミング言語を学ぶにしても、そこで共通的に通用する大切な事柄があります。それらを記述するコードとはどのようなものなのか。生徒に語りかけられる内容をできるだけ盛り込みながら、プログラミングのコードを本質的に捉えようとする試みます。

2 プログラミングを学習するということ

本来プログラミングは、それを教える教科、高校だと情報科の範疇だけで完結して捉えるべきものではありません。学習活動のすべての場面で、その能力が遺憾なく活用されることに越したことはないからです。しかしその足場を、とりあえず情報科に求めるしかないのだったら、私たちはその教育の枠組みの中で考えざるをえません。

誤解を恐れずに大雑把にいうと、情報科の学習は次の要素で構成されていると考えられます。

- 情報の基礎
  - コンピュータやネットワークの仕組みの理解した上での、情報というものへの理解
- 問題解決への応用
  - コンピュータやネットワークを活用した問題解決（データベースを含む）
  - モデル化とシミュレーションによる問題解決
  - プログラミング（アルゴリズム）による問題解決

「情報の基礎」はそれとして別に考えることにして、「問題解決への応用」を効率的そして効果的に学習するためにはどうすればよいでしょう。そこで有効であると思われる方法の一つが、「問題をモデル化して、プログラミングよってシミュレーションし、段階的に（スパイラルに）その解決に近づく」というアプローチです。そこで扱う問題が、身近なものであったりネットワーク上のものであったりすることで、「問題解決への応用」についての学習のほとんどを網羅することができます。

問題をモデル化して表現したり、解決への道筋をアルゴリズムとして表現したりするためには、シンプルなコードでプログラムを記述できなければなりません。その意味で Python は、純粋にアルゴリズム的な考え方に触れようとしたときに最適なプログラミング言語です。

プログラミングのコードは言語の欠片です。その欠片を紡いで文脈を生成する行いは言語活動です。コンピュータに文脈を伝えるために試行錯誤する行いはアクティブな学びです。このように考えると、プログラミングは総合栄養満点的な学習活動になり得るとは思いませんか。

今回のワークショップでは、お馴染みの遊び「双六（スゴロク）」をテーマに取り上げ、どのようにしたら遊びに耐え得るレベルにまでゲーム化できるかを考えます。基本的なルールをアレンジしながら、試作品（プロ

[*1] プログラミングを学ぶことで、「モデル化とシミュレーション」という、情報科における一つの大きな分野をも合わせて網羅できるということである。

[*2] 実験や経験を積み重ねながら、次第に求める結果に近づこうとする方法であり態度。

[*3] コードは言葉の断片であり、文脈の中でそれらに意味を与え、さらに組み合わせることで物語を紡ごうとする活動は、まさしく言語活動そのものではないか。

[*4] 協働的に行う活動やディスカッションなどは確かに活動的な学習活動の一面ではあるが、最終的に求められるのは、各自が様々な思考や思索の活動を行うことで、自らの世界観を深め広げることである。プログラミングにおける試行錯誤の活動は、そうした意味においてまさしくアクティブラーニング（能動的な学習活動）そのものである。

[*5] 試作品のこと。まずは手始めとして作ってみて、段々改良していくための原型となるもの。

トタイプ）から始まり、最終的には楽しく遊べる程度のゲームを作成しましょう。
覚えることはほんのわずかです。ロジックの部品であるシンプルなコード、Pure Code で遊んでみましょう。

3 Python でプログラミングをするためには

Python というプログラミング言語の処理系には、その開発の発展進化の歴史に沿って、主に Python2 系と Python3 系とがあります。Python2 系で書かれたプログラムがすでに世の中には大量にありますが、これから学習を始めようとするには、迷わずに Python3 系を選択すべきです。

本稿執筆時において、Linux や OS X にはあらかじめ Python2 系がインストールされています。ですから、Python3 系を利用するためには、Windows も含めて別にインストールの作業が必要です。基本的には Python の公式サイト[*6] で配布しているソースコードから自分でビルド[*7]したり、OS X や Windows でしたら、ビルド済みのパッケージを簡単にインストールしたりします。

その他にも、プログラミングするための開発環境が一緒になったパッケージ[*8]もあります。このような開発環境のパッケージは、通常はコンピュータにインストールしなければならないのですが、中にはインストールしなくても、USB フラッシュメモリなどのコピーしておいて、どこでも手軽に利用できるポータブルなものもあります。

Python で書いたプログラムを実行するには、次のような方法があります。

- Python 本体を起動した状態で、インタープリタを対話的に利用して実行する
- 独立した実行ファイルにして、コマンドとして実行する
- ファイルとして書いて Python から実行する
- 開発環境の中で、書いたプログラムを実行する

本稿ではポータブルな開発環境として PyScripter [*9] を利用します。では早速、PyScripter を起動して、Python プログラムのコードを書いて動かしてみることにしましょう。

4 どのようなプログラムを作ろうとしているのか

4.1 テーマを踏まえた方向性

本稿のテーマは「モデル化による仮説形成と、コードによる問題解決」というものです。概要としては、次のようなアプローチをすることになります。

ある事柄を抽象化（モデル化）してプログラムの中に形成する（オブジェエクト）。そして、そのプログラムを動作させる（シミュレーション）ことで、その事柄の本質に近づこう（問題解決）とする。最初は簡単な試作品（プロトタイプ）的なレベルであっても、試行錯誤を続ける（ヒューリスティック）ことで、最終的には完全ではなくても満足のいく領域に問題解決することができる。

4.2 何を扱うのか

プログラミングを活用した問題解決ですから、初めから難しいことを扱うことは大変です。ここでは、私たちに馴染み深い双六（スゴロク）を取り上げてみましょう。

最初に、シミュレーションを行う上で避けて通れない「偶然性」を扱えるようになりましょう。それには、乱数の扱いに慣れることです。乱数の働きを扱うためには、Python の基本パッケージでは不足があるので、「ライブラリをインポートする」ことを学びます。事の初めにインポートを学ぶのは、学習の順序性からいっ

[*6] https://www.python.org/
[*7] プログラムとして書いたソースコードから、コンパイルして機械語に変換したり、他のライブラリとのリンクなどを行い、最終的な実行可能ファイルを作成すること。
[*8] PyCharm, WinPython, Jupyter Notebook, Jupyter QtConsole, Spyder など
[*9] Google が開発しましたが、今は停止状態です。https://sourceforge.net/projects/pyscripter/

てどうなのか、という声もありそうですが、「だって必要なんだもん！」ということです。

その後は、とても短い道程（コース）を対象としたスゴロクのモデルを構築します。それには、簡単なルール「出た目の数だけ進む」を与えたプロトタイプを作成します。ただこれだけのことでも、「進んだことをどのように表現するのか」とか「進んだ位置をどのように保持するのか」など、プログラミングとして考える要素は少なくはありません。

その後は、プログラミングしたものがスゴロクらしく動作をすることを確認しながら、次第に高度なルールを付け加えていきます。

スゴロクは一度だけサイコロを振ればよいわけではありませんから、何度も繰り返す必要があります。とはいえ、いつかは終了する必要があります。上がり（ゴール）の判断が必要ですね。複数人で早く上がりに到達することで競うことを考えれば、繰り返した回数を使って判定することも考えられます。他にも、ある特定の目に進んだら、無条件にさらに何目か先に進むことができるとか、そこでは一回休みになるとか。そういえば、上がりにはちょうどの目が出た時にしか叶わない、などというルールもありました。

ゲームとしての完成度を考慮するのならば、ゲームが進行するプロセスや勝敗の結果を可視化できるというのも大きな魅力です。このように高度なレベルまでいっぺんに辿り着くことはもちろん、それを初めから想像することは難しいことです。しかし、プロトタイプから始まって少しずつ完成品に近づくというアプローチでしたら、試行錯誤を楽しみながらやり抜くことが可能です。

4.3 プログラミングの基本の基本

プログラミングの基本構造は、大きな括りとして「**入力**したものを**処理**して、結果を**出力**する」というものです。

数や文字（列）といった値（データ）を扱うために、変数に代入[*10]しておくことが入力です。データは、プログラムの中に直接書いて変数に代入することもあれば、キーボードやファイルから読み込んで代入することもあります。

変数を使っていろいろな計算を行うことが処理です。処理には単純な代数的計算だけではなく、扱うデータの型[*11]によって様々なものがあります。あまりにも膨大な処理の方法がありますが、目的と必要に応じて、少しずつ学んでいけばよいことです。

そして、その計算結果をユーザが確認できるようにすることが出力です。通常は、関数 print() などを用いてディスプレー（モニター）に出力しますが、プリンターに印字したりファイルに書き出したりすることも出力に当たります。

本稿では、プリンタやファイルについては扱いませんが、ここでの学習を終えた暁には、きっとファイルの扱いはどのようにすればいいのだ！といった意欲が湧いていることと思います。余力があれば、ぜひ触れてみたいものです。

5 コア・チュートリアル

5.1 サイコロを振って、出た目の数を調べる

5.1.1 サンプル・コード - sugoroku01.py

```
# 乱数を使うための下準備
from random import randint

# サイコロを振る（0 から 5 までの値を乱数として得る）
```

[*10] 「代わりに入れる」というよりも、「変数の名前に、値を保持しているメモリの番地を割り当てる」というのが正しい捉え方です。かえって、難しい？

[*11] 数値型、文字列型、リスト型、タプル型、辞書型などがあります。

```
# 変数に値を「入力」
me = randint(0, 5)

# サイコロの目を表示する
# 変数の値を「出力」
print(me)

# これまでの一連の処理は「逐次処理」
```

5.1.2 コメントを書く

サンプルコードの中で、記号#がついた行は、コメント行です。コメント、つまり注釈は、プログラムの実行には何も影響を及ぼしません。記号#は行頭に置かなくてもよく、行の途中にあった場合には、その#から後ろ行末までをコメントとして解釈します。

Python のようなプログラミング言語は、インタープリタ型[*12]の言語と呼ばれています。interpreter とは通訳者のことで、書かれたプログラムのコードを最初から 1 行ずつ読み込んでは実行していきます。そのときに、コメント行は実行せずに飛ばして進んでしまいます。

本稿にあるサンプル・コードの中にも、多くのコメントが書かれています。新規に書かれたコードには、基本的にコメントが付いています。以前にコメントが付いたコードには、2 度目からはコメントが付いていません。コメントがなくても、そのコードの意味が理解できるようになっていることが理想です。最初は行きつ戻りつ、分からなくなったら以前のコメントを参照して読み進めてください。

5.1.3 コードの中の、空白の扱い

1 行目にある from random import randint で分かるように、「半角の空白（文字)」は一般的に、コードの断片の区切りを明確にするために用います。日本語でいうところの「分かち書き」のようなもの、というか、英文としてみたときにも当たり前のことです。

ところで、me = randint(0, 5) のようなコードではどうでしょうか。me=randint(0,5) と書いてもよさそうですが、等号=の両側やカンマの後に半角の空白が 1 文字ずつ挿入されています。

これは、単にコードの読みやすさを考えて挿入したものです。Python インタープリタはありがたいことに、こうした半角空白を無視して解釈しますが、半角空白が入ることで読みにくいと感じるのならば挿入する必要はありません。[*13]

5.1.4 モジュールのインポート

乱数を扱うための関数などは、Python にもとから組み込まれているものではありません。[*14]Python 本体は、物事は「なるべく小さく単純なものの組み合わせであるべき」という思想がそこにはあります。必要なものは、随時外から持ってきて付け加えながら使えばよいと考えているのです。

[*12] 他にも、Ruby や Perl などもインタープリタ型のプログラミング言語です。一方に、コンパイラ型のプログラミング言語があります。compiler は翻訳者です。プログラムのコードを一括して読み込んで、コンピュータが直接実行できる機械語に翻訳してしまいます。機械語を実行させるので、インタープリタ型の言語より高速に動作するプルグラムを作れます。ちょっと敷居は高いですが。この仲間には C がありますが、Java などはその中間に位置するような中間言語型といえます。中間言語にコンパイルすれば、どんな OS でも Java 仮想マシンがあれば、その上で動作します。

[*13] プログラミングに不慣れな人は、こうした空白を入れることを嫌って、ビチビチに詰まったコードを書くことが多いようです。確かに、どのようなときに半角空白を挿入したらよいのかというルールみたいなものが厳然とあるわけでもないので、そうした恐れを抱く気持ちはよくわかります。しかし、コードが読みやすいということは、間違いを防いだり、改善改良のために後から読んだり、他の人が参考にしやすかったりと、そのメリットは計り知れません。ぜひ試行錯誤をして、自分なりのコーディング・スタイルを持つべきだと思います。

[*14] 関数 print() などのように、何の手間もなくすぐに使えるものは、Python のもともと組み込まれているものです。こうしたものを、組み込み関数と呼びます。

乱数を扱うには、random というライブラリ[*15]から、randint という関数[*16]を読み込みます。

5.1.5 代入記号 =

me = randint(0, 5) という式は、どのようなことをしているのでしょう。

A = B という形式は、数学では等式などと呼んでいます。つまり、A と B との値が等しいことを表しています。ところが数学では、「x の値を 2 とします」などといって、x に 2 を代入することを x = 2 などと等式の形式で書くのです。よく考えてみればこれはおかしな話で、両辺が等しいことを表す「等式」と、左辺の変数に右辺の値を代入するときの「代入式」とは区別されるべきものです。数学において人間は、文脈に応じて等式と代入式とを区別できているのですね。

一方で、コンピュータにはそのような文脈を判断するようなことは一般的にはできないので、プログラミングでは両者を区別して書くことが多いのです。

A = B という形式の式は、B の値を A の値として代入することを意味します。右から左への方向性を持って代入を考えます。等式を表す場合には、多くのプログラミング言語では A == B などと書きます。

5.1.6 処理とは、この 3 種類

プログラムで処理を記述する場合、その処理には大きく考えて 3 種類があります。[*17]

このサンプルコードでは単純な逐次処理をするのみですが、今後に備えて処理の種類を以下にまとめておきます。それぞれの詳細については、サンプル・コードに中に出てきたときに説明します。

1 つ目は「逐次処理」です。これは何も難しいことではなく、プログラムを書いた順に実行処理します。とても単純な処理の流れですが、もちろんこれだけで複雑なプログラムを作成できるわけではありません。

2 つ目は「条件処理」です。条件が成り立つ場合と成り立たない場合に応じて、それぞれの処理を分けて書く必要があります。「条件の成り立つ場合には、さらに別の条件があって…」などのように、入れ子状態[*18]に複雑な処理になることがあります。

3 つ目は「反復処理」です。同じ処理を繰り返すのですが、一定回数だけ繰り返すとか、ある条件が成り立つ（成り立たなくなる）まで繰り返すなどの種類もあります。コンピューターは、この繰り返しを高速に行うことができる疲れ知らずでしたね。

5.1.7 ファイルの保存と文字コード

今作成している Python のプログラムを、ファイルに保存しておきましょう。ファイルの名前は、本稿に沿って Python を学習するのでしたら、サンプル・コードに示しているように sugoroku01.py としておきましょう。ファイル名のドット「.」以降の部分を拡張子[*19]といって、そのファイルがどのような種類のものであるのかを示します。ここでの py はもちろん、Python のプログラム・ファイル[*20]であることを示しています。

注意しておかなければならないことは、ファイルを保存するときの文字コードです。Python はインターネットとの親和性が高いので、文字コードとしては UTF-8 を扱うのが既定（デファルト）となっています。ところが、PyScripter は Windows で動作しているので、そのデフォルトは Ansi (Shift-JIS) になっているよ

[*15] クラス（本稿では扱っていないので、読み飛ばしてください。）や関数の定義してまとめたものをライブラリといいます。また、ライブラリの実態であるファイルとその保存の仕方を指してモジュールということがあります。

[*16] クラスや関数などを一括して、Python はオブジェクトと呼んでいます。何でもかんでも Python が扱うもの、つまりデータはオブジェクトである、と考えても差し支えないと思います。

[*17] 言い換えると、この 3 種類で全ての事象を記述できるということです。考えてみましょう、あなたの毎日の生活を。物事を逐次的に行い、そのことを毎日のように繰り返し、ときには条件によってやることを変更する。そのような説明の仕方で、どんな人生をもプログラミング言語で記述できるのです。意外を人生は、単純なんだなあ…。

[*18] ある論理構造の中に、さらにその論理構造があるような場合、それを入れ子 (nest) 状態、あるいは構造であるといいます。この入れ子構造が何段にも深くなり場合もあります。マトリョーシカみたいなもの。

[*19] 例えば、.docx はワードプロセッサである Word の、.js は JavaScript で書かれたファイルです。

[*20] プログラミング言語で書かれたファイルを、一般的にはプログラム・ファイルと呼びます。しかし、コンパイラ言語のようにドーンと一括してマシン語に変換してしまうのではなく、1 行ごとに変換しながら実行するインタープリタ言語では、それをスクリプト・ファイルと呼んだりすることがあります。script とは台本のことですね。どちらにしろ、書かれた記号列のことを、コードと呼びます。

うです。メニュー「編集」から「ファイルフォーマット」を辿り、デフォルト値を確認した上で UTF-8 に設定しておいてください。[*21]

5.2 サイコロを振って、出た目の数だけ進む

5.2.1 サンプル・コード - sugoroku02.py

```
from random import randint

# 位置を表す変数を準備（初期化）する
ptr = 0

me = randint(0, 5)

# 出た目の数だけ進む
# 変数の値のインクリメント ptr = ptr + me
ptr += me

# 出た目と現在の位置とを表示する
print(me, ptr)
```

5.2.2 変数の初期化

多くのプログラミング言語では、変数を利用するときには利用する前に、その変数を「使いますよ」と宣言します。例えば他のプログラミング言語では、var hensu とか int hensu などと書きます。[*22]

Python には、そうした宣言は必要ありません。使いたいときに使い始めればよいのです。しかし、変数として名前を付けたときには必ず値が代入されなければなりません。値としての実体のない名前は、Python が動作することには受け入れられないのです。[*23]

値を持たない変数名を使おうとすると、Python はエラーを検知して実行を止めてしまいます。これはよく起こるエラーなので、エラー発生の原因の一つとして頭の中に留めておくとよいかもしれません。

勝手な思い込みで変数を使うと、後から思いも掛けないエラーが紛れ込むことがあります。一般的に、変数を利用しようとするときには、その最初の値をきちんと代入しておくことが望ましいです。つまりこれは、「変数は初期値をきちんと代入（初期化）してから使いましょう」という鉄則といえるものです。

5.2.3 値のインクリメント

ある変数の値を増加させる、あるいは減少させる[*24]ことが必要になる場合があります。例えば、変数 hensu の値を 2 だけ増やそうとするとき、hensu = hensu + 2 と書きますが、この意味はすぐに分かりますか？

思い出してください。記号 = は、右辺から左辺への代入記号でした。

右辺を見ると hensu の値と 2 が足し算されています。もし、最初の hensu の値が 5 だとすれば、この右辺の足し算によって、右辺は 7 の値を持つことになります。そしてこの値が、代入の動作によって、左辺の値になるわけです。左辺は hensu でしたから、元の 5 の値に変わって新しく 7 の値が代入されるわけです。結果として、hensu の値は 5 から 7 に、2 だけ増加したことになります。

[*21] もし、このことを忘れて Ansi のままに保存してしまうと、次の機会に PyScriper を使って該当のファイルを開いたときに、残念ながら日本語のコメントなどは文字化けしてしまって読むことができません。動作には支障がないのですけれども。

[*22] var とか int を付けることでデータ型が決まります。このことで、その変数を使用するためのメモリ領域が確保されます。

[*23] つまり Python では、変数にデータが割り当たった段階で初めて、変数の名前を管理するメモリ領域と、データの値が記憶されるメモリ領域とが紐付けされて意味を持つのです。

[*24] 増加させることをインクリメント (increment) する、減少させることをデクリメント (decrement) するといいます。

このような変数の値の増加や減少の操作は、プログラミングには頻繁に出てきます。そこで、短縮した書き方が用意されていて、Python ではこのような場合、hensu += 2 と書くことができます。[*25]hensu という変数名を 2 度も書かなくて済む、ということですね。

5.2.4 関数 print()

print(hensu) と書けば、hensu の値をディスプレーに表示（出力）します。関数 print() は、いろいろな使い方ができる関数です。今回のように、print(hensu1, hensu2) のように変数をカンマで区切って並べると、それぞれを半角空白 1 個で区切って出力します。

他にもある使い方については、これから随時紹介することにします。

5.3 サイコロを何度も振って、出た目の数だけ進む

5.3.1 サンプル・コード - sugoroku03.py

```
from random import randint

ptr = 0

# とりあえず 20 回くらい繰り返す（反復処理）
for i in range(20):
    me = randint(0, 5)
    ptr += me
    print(me, ptr)
```

5.3.2 反復処理 for i in range()

ある処理を 10 回繰り返して行おうとするとき Python では、回数をカウントする適当な変数（ここでは i）を使って次のように書きます。[*26]

```
for i in range(10):
    処理
```

関数 range(n) は、0 から始まり n-1 までの、合計 n 個の整数を生成します。[*27]for 文は、この range(n) の中から順に整数を取り出して、変数 i に代入して処理を行います。[*28]つまり最初は、i = 0 として処理を行い、次に i = 1 として同じ処理を行い、最終的には i = 9 として処理を行ったら反復処理を終えます。[*29]

このときに 2 つの注意があります。

- for 文の末尾には記号コロンを書く。このことで、次の行からには処理の内容が書かれていることを認識する。
- 反復する一連の処理は、それらの文頭を同一のレベルに字下げ（インデント）して、処理ブロックとして表す。

[*25] 1 だけ増やすといいことは、何かの回数をカウントする場合などによく出てくる処理です。このときには本来、hensu = hensu + 1 と書くところを、hensu += 1 と書けるというわけです。Pyhton では書けませんが、他の言語ではこれを、hensu++ と書けるものもあります。馴染めますか？

[*26] 他のプログラミング言語を学んでいる人は、for(i=0, i¡10; i++){ 処理 } のような書き方を知っているかと思う。

[*27] range(10) として生成される 0 から 9 までの連続したデータのことを、シーケンス (sequence) 型のデータであるといいます。

[*28] シーケンス型のデータから 1 つずつの要素を取り出して処理を繰り返す仕組みのことを、イテレータ (iteratoer) といいます。

[*29] もちろん、次の処理が書かれていたら、逐次的に次の処理に動作が移ります。

5.3.3 処理のブロック

反復処理に限らず、ひとまとまりの処理をブロックとして扱う場面は少なくはありません。例えば、このあとすぐに条件処理を行いますが、そこでもこの「コロンとインデント」による処理ブロックの書き方が使われます。

処理のブロックは、一般的なプログラミング言語では中括弧で括ることが多いのですが、「コロンとインデント」は Pyhon コードの特徴の一つです。本稿でのプログラムのコードは、インデントに半角空白を 4 個分を用いている。[*30]

5.3.4 反復処理のいろいろ

反復処理を行うには、ここで利用した for 構文以外にも、while 構文を使うことができます。

```
while 条件:
    処理
```

条件が成立している間は、処理を繰り返します。

for 構文では、反復の回数はあらかじめ見積もることができます。while 構文では、条件によって反復回数が変わってしまうので、その回数を見積もることができません。

スゴロクのモデルでは、サイコロを振る回数を最初から想定しておくことはおかしいのですが、イテレータの考え方を用いる for 構文を紹介したかったので、本サンプル・コードでは大雑把に回数を見積もってしまっています。

5.4 上る（ゴール）ことができるようにする

5.4.1 サンプル・コード - sugoroku04.py

```
from random import randint

# ゴールの目の位置を決める
ME_MAX = 9

ptr = 0

for i in range(20):
    # 目を 0, 1, 2 に限定してサイコロを振る（今後のゲーム性を考慮）
    me = randint(0, 2)
    ptr += me
    print(me, ptr)

    # ゴールに到達したか判定する（条件処理）
    if ME_MAX < ptr:
        # for ループ（反復処理）を抜ける（結果的にプログラムは終了）
        break
```

[*30] インデントに半角空白を 4 個分を用いることは、Python 自身が推奨している。他のプログラミング言語では、インデントにタブ（Tab コード）を用いることが多い。タブの間隔は一般的に、利用するエディタの設定によってさまざまである。

5.4.2 定数

ゴールの目の位置を最初に決めたが、これはプログラムの中で変更されることのない定数です。逆にいうと、プログラムの実行途中で変更されてしまうと、プログラム全体の設計が狂ってしまいます。[*31]

このような定数を表す場合の変数名には、アルファベットの大文字を使います。これは、他のプログラミング言語でも同様の作法と言えます。

5.4.3 条件処理 if elif else

条件によって処理の内容を変更する場合に、if 文を用います。基本的な使い方は以下の通りです。

○ A するか、しないか

```
if 条件 1:
    処理 A
```

○そうなら A する、そうでなければ B する

```
if 条件 1:
    処理 A
else:
    処理 B
```

○そうなら A する、ああなら B する、さもなければ C する

```
if 条件 1:
    処理 A
elif 条件 2:
    処理 B
else:
    処理 C
```

elif は必要なだけ書けるので、条件によって分岐する処理が多くあっても対応できます。[*32]また、if を用いた構文は入れ子状態にできるので、ある条件のもとでさらに条件を考慮しなければならないなど、複雑な条件処理も可能です。[*33]

5.4.4 0〜3 の目しかないサイコロ

スゴロクをモデル化したゲームを作成しているのに、サイコロの目を制限するのはなぜでしょう。今ここでは、スゴロクがどのようなものであるかをモデル化して、シミュレーションによって確かめようとしています。このときに、サイコロの目が最大 5 であって、スゴロクの盤面をどんどん進んでいくことは本質的なことではありません。サイコロによって目を進むこと、1 回休みの目があること、ボーナス的に進める目があること、区切りよくゴールをするにはどうするかなど、スゴロクにとって本質的なものは進む目の大きさとは関係のないところにあります。

こうしたときには、サイコロの目の大きさを制限することでアルゴリズムを簡潔にすることを心掛けましょう。モデル化とシミュレーションの手法として大切なことだと思ます。

[*31] もちろん、ゴールの目の位置を変更すると別のゲームになってしまう。この変数の値は、ゲームのあり方を決定するし、いわばゲームの個性を表したものです。

[*32] 他のプログラミング言語には、多くの同等の条件による分岐が必要になった場合に、switch〜case のような便利な構文があるが、Python にはないのはなぜだろう？

[*33] もっとも、プログラムを書く側にとっても読む側にとっても、それが読みやすいものであるかどうかは、また別の話です。

5.4.5 反復処理からの脱出

サイコロを 20 回も振れば、とりあえずは上がりに辿り着けるだろうという目論見で反復処理を行っています。しかし、20 回の試行を行わないうちに上がりに辿り着くことができたのなら、残りの回数の試行は無意味であるし、労力[*34]と時間の無駄でしょう。

そこで、上がりに辿り着いたら反復処理から抜け出すことを考えました。それを行うのが break 文です。反復処理の繰り返し（ループ loop）から脱出して、動作をその次の処理に移行します。

5.5 上がるまで、サイコロを何回振ったかを調べる

5.5.1 サンプル・コード - sugoroku05.py

```
from random import randint

ME_MAX = 9
ptr = 0

# 振った回数を表す変数を準備（初期化）する
times = 0

for i in range(20):
    me = randint(0, 2)
    # 振った回数をインクリメントする
    times += 1
    ptr += me
    # 振った回数、出た目、現在の位置を表示する
    print(times, me, ptr)

    # ゴールの判定
    if ME_MAX < ptr:
        break
```

5.5.2 サイコロを振った回数

これまでは、1 人遊びのようなものであったが、これからは複数人で競い合って遊べるような方向へ進みましょう。サイコロを振った回数を記憶しておいて、上がった段階での回数が少ない者を勝者とすることにします。

変数 times の値を 0 に初期化して利用しています。回数のインクリメントは 1 ずつであるから、times += 1 と書くことができます。このような表現は頻繁にでてくるので、読み書きに違和感のないようになりましょう。

5.6 特定の目の位置に来たら、ボーナスで進む

5.6.1 サンプル・コード - sugoroku06.py

```
from random import randint
```

[*34] コンピュータは、この場合の労力を厭わないとは思うがそれにしても、ですね。

```
ME_MAX = 9
ME_BONUS = 5    # ボーナスを得る目の位置
STEP = 2                # ボーナスで進む目の数

ptr = 0
times = 0

for i in range(20):
    me = randint(0, 2)
    times += 1
    ptr += me
    print(times, me, ptr)

    if ptr == ME__BONUS:    # 進んだ目が 5 ならば
        ptr += STEP                 # さらに STEP だけ進む
        print(times, me, ptr)

    if ME_MAX < ptr:
        break
```

5.7 上がりの目を過ぎたら、元に戻る

5.7.1 サンプル・コード - sugoroku07.py

```
from random import randint

ME_MAX = 9
ME_BONUS = 5
STEP = 2

ptr = 0
times = 0

for i in range(20):
    me = randint(0, 2)
    times += 1
    ptr += me
    print(times, me, ptr)

    if ptr == ME_BONUS:
        ptr += STEP
        print(times, '+', ptr)

    if ME_MAX == ptr:           # ゴールの位置にちょうど到達できたら
        break                   # ループ（反復処理）を抜ける（結果的にプログラムは終了）
```

```
    elif ME_MAX < ptr:          # ゴールの位置を過ぎてしまったら
        ptr -= me               # 進んだ目の分だけ元に戻る
        print(times, -me, ptr)
```

5.8 進んだ目の位置が偶数なら、1 回休み

5.8.1 サンプル・コード - sugoroku08.py

```
from random import randint

ME_MAX = 9
ME_BONUS = 5
STEP = 2

ptr = 0
times = 0

for i in range(20):
    me = randint(0, 2)
    times += 1
    ptr += me
    print(times, me, ptr)

    if ptr == ME_BONUS:
        ptr += STEP
        print(' ', '+', ptr)

    # 進んだ位置が、ゴールを超えておらず、かつ、偶数ならば（%は剰余の演算子）
    if  ptr < ME_MAX and ptr % 2 == 0:
        # 振った回数をさらに 1 回増やして、1 回休んだことにする
        times += 1
        print(times, '-', ptr)

    # ゴールの判定
    if ME_MAX == ptr:
        break
    elif ME_MAX < ptr:
        ptr -= me
        print(' ', -me, ptr)
```

5.8.2 剰余演算子 %

四則や累乗の計算ではなく、剰余演算はそれほど馴染みがないかもしれません。割り算したときの余りを剰余といって、例えば 20 を 3 で割ったときの剰余（余り）は 2 となります。[*35]

[*35] 高校の数学には、剰余定理というものがありました。

プログラミングでは、剰余計算を使ってデータをグループに分けることをよくします。[*36]例えば 40 人在籍するクラスの生徒を、3 つのグループに分けることを考えます。あなたなら、どのような方法をとりますか。

着席している生徒を見て、人数が同じくなるように線引きしますか。それとも、生まれた月を使って、1 月から 4 月、5 月から 8 月、9 月から 12 月のように分けますか。もしかして、座っている順番に、グループ 1、グループ 2、グループ 3、グループ 1、…のように繰り返して分けるかもしれませんね。

剰余計算は、この最後の方法を計算でするようなものです。3 グループに分けるので、生徒番号を 3 で割って余りを計算すると、それは 0, 1, 2 のどれかになります。この値によってグループに分けるのです。

本サンプル・コードのように、ptr % 2 == 0 を判断するということは、結局のところ、偶数か奇数かを判断していることになります。

5.9 進んだ位置が偶数でも、出た目が 0 なら（かわいそうだから）休みにしない

5.9.1 サンプル・コード - sugoroku09.py

```
from random import randint

ME_MAX = 9
ME_BONUS = 5
STEP = 2

ptr = 0
times = 0

for i in range(20):
    me = randint(0, 2)
    times += 1
    ptr += me
    print(times, me, ptr)

    if ptr == ME_BONUS:
        ptr += STEP
        print(' ', '+', ptr)

    if  ptr < ME_MAX and ptr % 2 == 0:
        if me == 0:
            # pass 文は、何もしない。インデント構造を維持するために必要。
            pass
        else:
            times += 1
            print(times, '-', ptr)

    if ME_MAX == ptr:
        break
    elif ME_MAX < ptr:
```

[*36] 大学で学ぶ高等数学には、剰余類という考え方があります。

```
        ptr -= me
        print(' ', -me, ptr)
```

5.9.2 何もしない、という処理 pass

処理を何もしないのに、そのことを表す処理 pass が存在するということには、いったいどのような意味があるのでしょうか。本サンプル・コードで、もし pass の記述がなかったら、プログラムコードはどのようになってしまうのでしょうか。

pass を記述しないと、きっとエラーが出てしまって動作が止まるのではないでしょうか。その理由は、if 構文の形にあります。

if とそれに続く条件の後には、必ずコロンがきます。（このことは問題ないですね。）そしてその後には、条件が成立した場合の処理（ブロック）がインデントされて書かれるはずです。ここで、何も処理がいらないからといって、pass がなければどのような状態になるでしょう。そうですね、処理ブロックが見当たらないことになり、これでは文法的に困ったことになります。

文 pass は、数学の集合でいうところの、空集合を表すようなものです。集合としての枠組み話あるのだけれど、中身の要素が何もない (pass) という状態を表しているのです。

5.10 複数人で交互にサイコロを振り、ゴールへの回数の少なさを競う

5.10.1 サンプル・コード - sugoroku10.py

```
from random import randint

ME_MAX = 9
ME_BONUS = 5
STEP = 2

ptr = 0
times = 0
STEP = 2

# キーボード（標準入力）から入力を促すためのメッセージ
message = 'あなたの番になったら [OK] ボタン、あるいは [Enter] キーを押してください。途中でいやになたら、[Q] を入力してください。'

for i in range(20):
    # キーボードから入力を促し、変数 ret に代入する（名前を付ける）
    ret = input(message)
    # 'q' とか'Q' が入力されたら
    if ret == 'q' or ret == 'Q':
        # ループから抜けて、結果としてプログラムを終了する
        break
    # このプロセスによって、プログラムはユーザの意志によって逐次的に動作する

    me = randint(0, 2)
    times += 1
    ptr += me
```

```
    print(times, me, ptr)

    if ptr == ME_BONUS:
        ptr += STEP
        print(' ', '+', ptr)

    if  ptr < ME_MAX and ptr % 2 == 0:
        if me == 0:
            pass
        else:
            times += 1
            print(times, '-', ptr)

    if ME_MAX == ptr:
        break
    elif ME_MAX < ptr:
        ptr -= me
        print(' ', -me, ptr)
```

5.10.2 キーボードからの入力

キーボードからの入力には、関数 input() を用いるが、ここで、「5」が刻印されたキーを押したときのことを考えてみよう。例えば、x = input('what?') などとして入力を受け取った変数 x を使って、y = 10 * x などと演算が可能なのか、という問題である。

関数 input() がキーボードから受け取るデータは、すべて文字（列）型として扱われます。少し考えればわかりますが、「5」が刻印されたキーが押されたとしても、それが数値なのか文字なのかはその段階で判断ができないからです。

もし数値として扱いたい場合には、文字を数値に変換する必要があり、それは関数 int() を用いて行います。もっとも、アルファベットのキーが押されたのにそれを数値に変換するには無理がありますし、そこではエラーが発生してしまいます。本来は、押されたキーが数字なのか文字なのかを判断してから、適切な処理をするようなコードを書く必要があります。

5.10.3 嫌になったら止める、ということ

一般的に、物事は始めるよりも、止めることの方がむずかしことの方が多いです。アプリケーション・プログラムも、アイコンをマウスでダブル・クリックすると開始しますが、終えるためにはそれなりの処理が必要です。自分で作成したプログラムも、始めたはいいが止めるためには、まだまだ何回もクリックしなければならない、なんていうこともよくあります。

動き出したプログラムを気持ちよく終えるためにも、終了の仕方にはよく考えた方法を採用しましょう。

5.11 ゲームの進行を分かりやすくする

5.11.1 サンプル・コード - sugoroku11.py

```
from random import randint

# ゲームのプロセスを可視化するための出力を関数化する
# 引数は status、戻り値はない（正しくは、オブジェクト None が return する）
```

```
def process(status):
    if status == 'Walk':
        print(times, me, ptr)
    elif status == 'Jump':
        print(' ', '+', ptr)
    elif status == 'Skip':
            print(times, '-', ptr)
    elif status == 'Back':
        print(' ', -me, ptr)

ME_MAX = 9
ME_BONUS = 5
STEP = 2

ptr = 0
times = 0

message = 'あなたの番になったら [OK] ボタン、あるいは [Enter] キーを押してください。途中でいやになたら、[Q] を入力してください。'
# プロセスの状態を表す変数の初期化（空文字）
status = ''

for i in range(20):
    ret = input(message)
    if ret == 'q' or ret == 'Q':
        break

    me = randint(0, 2)
    times += 1
    ptr += me
    process('Walk')

    if ptr == 5:
        ptr += STEP
        process('Jump')

    if  ptr < ME_MAX and ptr % 2 == 0:
        if me == 0:
            pass
        else:
            times += 1
            process('Skip')

    if ME_MAX == ptr:
        break
    elif ME_MAX < ptr:
```

```
        ptr -= me
        process('Back')
```

5.11.2 関数についてあれこれ

関数は、よく使う、あるいは頻繁に使う処理を、名前を付けてまとめたものです。例えばこれまでよく使ってきた関数 print() は、頻繁に行う「データの値を出力する処理」のために定義され、素の状態の Python に元から組み込まれているものです。

関数の定義は、以下のような形式で行います。

```
def 関数名( 引数 ):
    処理
    return 戻り値
```

コロンとインデントによる処理ブロックの書き方は、これまでと同様です。

関数の定義は、def (define) の次に関数名を書き、丸カッコの中に、関数に与えるデータ（引数）を書き入れます。引数が必要ないときには書かなくてよいですし、引数が複数必要なときには、カンマで区切って書き並べることができます。

処理が終わったら関数が呼び出された（使われた）ところに戻りますが、そのときに処理の結果のデータを持ち帰る場合には、return の後にその値（戻り値）を書きます。この戻り値も、必要がなければ書かなくても構いません。関数 print() は、この手の関数です。

5.12 ゲームの進行を分かりやすくする

5.12.1 サンプル・コード - sugoroku12.py

```
from random import randint

# ゲームのプロセスの可視化を工夫する
def process(status):
    if status == 'Walk':
        print("{0:0>2}".format(times), "{0:>2}".format(me),  "{0:>2}".format(ptr-me), '
→', "{0:>2}".format(ptr), status)
    elif status == 'Jump':
        print('  ', '  ', "{0:>2}".format(ptr-STEP), '→', "{0:>2}".format(ptr), status)
    elif status == 'Skip':
        print("{0:0>2}".format(times), '  ', '  ', '  ', "{0:>2}".format(ptr), status)
    elif status == 'Back':
        print('  ', "{0:>2}".format(-me),  "{0:>2}".format(ptr+me), '→', "{0:>2}".format(ptr), stat

# 変数の初期化を 1 行に書く。
#（こんなふうにも書けますよ、という例であり、意味の異なる変数をこのように列挙するのはどうかと思う）
ME_MAX = 9; ptr = 0; times = 0; STEP = 2; status = ''

message = 'あなたの番になったら [OK] ボタン、あるいは [Enter] キーを押してください。途中でいやになたら、[Q] を入力してください。'

for i in range(20):
```

```
    ret = input(message)
    if ret == 'q' or ret == 'Q':
        break

    me = randint(0, 2)
    times += 1
    ptr += me
    process('Walk')

    if ptr == ME_BONUS:
        ptr += STEP
        process('Jump')

    if  ptr < ME_MAX and ptr % 2 == 0:
        if me == 0:
            pass
        else:
            times += 1
            process('Skip')

    if ME_MAX == ptr:
        break
    elif ME_MAX < ptr:
        ptr -= me
        process('Back')
```

5.12.2 書式付きの関数 print()

サンプル・コードにあるように関数 print() は、書式を用いた書き方をすることで、相当の表現力を持っています。また、出力を制御するような仕組みも兼ね備えています。本稿ではそのすべてを紹介しきれませんが、ぜひ各自で調べてみてください。

5.12.3 Python ならではのコード

本サンプル・コードでのセミコロンの使い方は、あまりお勧めできるものではありません。しかし Python には、他のプログラミング言語にはない（あるいは、あることはありますが）独特のコーディングもあります。

例えば、変数 a と b との値を交換する場合には、どのようなコードを書きますか？普通であれば、一時的な変数 x を用意して、

```
x = a    # a の値を x に退避（保存）する
a = b    # 安心して、a には b の値を代入する
b = x    # 退避（保存）しておいた a の元の値 x を、b に代入する
```

のようなコードを書きますが、Pythonhaha は次の 1 行のコードで済んでしまいます。

```
a, b = b, a
```

5.13 ゲームの進行をビジュアルに可視化する

5.13.1 サンプル・コード - sugoroku13.py

```
from random import randint

def process(status):
    if status == 'Walk':
        print("{0:0>2}".format(times), "{0:>2}".format(me),  "{0:>2}".format(ptr-me), '
→', "{0:>2}".format(ptr), status, end='')
        histogram(ptr)
    elif status == 'Jump':
        print('  ', '  ', "{0:>2}".format(ptr-STEP), '→', "{0:>2}".format(ptr), status, end='')
        histogram(ptr)
    elif status == 'Skip':
        print("{0:0>2}".format(times), '  ', '  ', '  ', "{0:>2}".format(ptr), status)
    elif status == 'Back':
        print('  ', "{0:>2}".format(-me),  "{0:>2}".format(ptr+me), '→', "{0:>2}".format(ptr), stat
        histogram(ptr)

# ゴールまであとどれくらいかを可視化する関数
def histogram(ptr):
    chara = '*'
    if ptr == ME_MAX:
        print(chara * ME_MAX + '|')
    else:
        print(chara * ptr + ' ' * (ME_MAX - ptr) + '|')

ME_MAX = 9; ptr = 0; times = 0; STEP = 2; status = ''
message = 'あなたの番になったら [OK] ボタン、あるいは [Enter] キーを押してください。途中でいやに
なたら、[Q] を入力してください。'

for i in range(20):
    ret = input(message)
    if ret == 'q' or ret == 'Q':
        break

    me = randint(0, 2)
    times += 1
    ptr += me
    process('Walk')

    if ptr == ME_BONUS:
        ptr += STEP
        process('Jump')
```

```
if  ptr < ME_MAX and ptr % 2 == 0:
    if me == 0:
        pass
    else:
        times += 1
        process('Skip')

if ME_MAX == ptr:
    break
elif ME_MAX < ptr:
    ptr -= me
    process('Back')
```

5.13.2 データを可視化する

数量的なデータを可視化するのなら、文字列の掛け算で簡単に実現できます。'z' * 5 とすれば、'zzzzz' という文字列を得ることができます。つまり、文字列と数値の掛け算は、文字列の反復を生むのです。データ型によって、それぞれに様々な演算が定義されているので、学習の進み方に合わせていろいろ吸収していくとよいでしょう。

本サンプル・コードでは、ヒストグラムの作成にこの機能を、関数 print() と共に用いています。

5.13.3 関数 print() の行末制御

関数プリントは、デフォルトでは出力の後に改行をします。これは、引数のオプション end に改行が指定されているからです。

改行をしたくない場合もあります。そのときには、end に対して「何もしてくれるな」とお願いすればよいのです。本サンプル・コードでそのことを確認してみてください。

6 乱数を扱うことでする授業

6.1 コンピュータが扱う乱数

本稿では、乱数発生を、スゴロクをモデル化してプログラミングすることでゲーム化するために、サイコロを振るという行為の肩代わりとして利用しました。サイコロは本来、どのような目が出るのか予想がつかないものです。

ところが実際には、サイコロの形にも微妙な歪みがあったり、その重心の位置にわずかな偏りがあったりします。真に予想のつかない乱数を、サイコロで発生させることには物理的にというか、現実的には無理があるものです。

プログラミングで扱う乱数に関する関数は、本当にランダムな数を発生しているわけではありません。一般的に、コンピュータで発生させる乱数は擬似乱数と呼ばれ、ある特定の法則によって生成されます。その法則を素となるのが seed といわれており、seed に関するメソッド（ここでは関数といって良いでしょうか）を使うとわかります。

random モジュールの seed というメソッドは、この擬似乱数の数列を初期化します。ですから、乱数を生成する仕組みを初期化すると、いつも同じ数列となって乱数が返ってきます。乱数とはいえ、本当に乱数なのかという点で、このことは頭に入れておくべき内容です。

普段は、seed をいちいち設定することはしないですね。その代わり、Python のライブラリが、その時の時刻を seed として乱数生成の仕組みを初期化してくれます。だから私たちは、ランダムな数が得られているよ

うに思えているわけです。

random モジュールには、このほかにもいろいろなメソッド（関数）があります。指定した間隔の間から、満遍なくランダムな数を一様に得ることができる乱数を、一様分布乱数といいます。その他にも、特定の分布の形に従う乱数を得ることができるメソッドもあります。[*37]

6.2 乱数はほんとうに乱数なのかを調べる

上の節で説明したように、プログラミングによって発生させる乱数は、それがほんとうに乱数なのかと問われれば、少しばかり不安は残ります。しかし、乱数を学ぼうとすることに対しては、間違いなく役に立ちます。

例えば、0 から 9 までの乱数を 10 個発生させて、関数 print() で出力したそれらを眺めます。10 個程度だと、出現した数には明らかな偏りを見出せるはずです。

次にそれらを 100 個発生させて観察します。100 個もあると、それらを目で追いかけるのは大変になりますから、何となく乱数であることを認めてしまいつつも、「本当に乱数なのかなあ」などと疑念も残るわけです。

これが 1000 個になるともうお手上げです。何とか乱数であることを、あるいは乱数とはいえないことを判断する方法が必要となります。さて、プログラミングの力を借りて、このことを検証する方法はないでしょうか。

6.3 乱数をプログラミングによって探求する

6.3.1 プログラミングの基礎と乱数

このことは、プログラミングの学習を始めるよいきっかけとなります。上の節で書いたことは、プログラミングの基礎的なことを踏まえることから始めると、ほぼ 1 授業時間単位に相当します。その後、本節にあるような内容を踏まえれば、数コマの授業時間に相当するでしょう。

6.3.2 乱数の出現頻度の集計

さて、まず、0 から 9 までの数がそれぞれ何回出現したかを計算する必要があります。0 から 9 までの 9 個の変数を用意して、乱数が発生するたびにそれらの変数をインクリメントしていくとよいでしょう。実は、本稿で扱っていないデータ型として、リスト構造があります。[*38]そのリストを利用すれば、反復処理を利用して効率的に集計できます。

6.3.3 出現頻度の可視化

集計した出現頻度の度数データは、それだけでは単なる数字の羅列なので、直感的に一様分布しているかどうかの判断はつきません。そこで、データの可視化が求められます。可視化の方法は、チュートリアルの中でも述べてきましたから、もうお分かりでしょう。ヒストグラムで使うキャラクタを決めておいて、それに度数を掛け算することで、その「キャラクタの連続」する量をヒストグラムの長さとするのでした。

100 個、1000 個と発生する乱数が多くなればなるほど、ヒストグラムも長くなり扱いづらくなりますから、5 個分とか 10 個分とかをまとめて扱うような工夫もすることができます。

6.3.4 統計的な検証

情報科の学習において、統計的な処理の方法を学ぶことは大きな目標の一つでもあります。この後は、出現頻度の平均を計算したり、分散や標準偏差を求めることで、客観的に一様分布であることを検証できます。

ここでの処理には、もちろん表計算ソフトを用いることも可能です。また、ここでの「乱数を扱う」ことが、表計算ソフトを使い始めることへの動機付けとなり、その使い方から学習を始めるというのも一つの方法です。

[*37] 統計には、正規分布やガウス分布などと呼ばれる分布があります。これらは、平均周辺の出現頻度がひじょうに高い乱数を生成します。このように、特定の統計的な分布に従う乱数も、Python は発生することができます。もっとも、他のプログラミング言語でも多かれ少なかれ、の話です。

[*38] 他のプログラミング言語では、配列などと呼ばれているものです。

さらに、表計算ソフトに統計的代表値を計算させるのではなく、Python を使って計算そのものをプログラミングでこなしてしまうこともできるでしょう。とにかく、乱数という分かっているようで漠然としていたものを、自分なりに明確に捉えようとする学習意欲をうまく引き出すことができれば、生徒のアクティビティによって学習の方向性が自然に見え始めるのではないでしょうか。

私には、そう思えます。

7　おわりに

高教研情報部会の研究集会、Python によるプログラミングのワークショップはいかがだったでしょうか。そして、今読んでくださっているこのテキストは、参考になりましたでしょうか。もし、ご意見ご感想をいただけるのでしたら、このテキストはアップデートしていきたいと考えていますので、どうか忌憚のないところをお寄せください。

そして、ほんの私事になりますが…

思い返せば、この情報部会の創設から係わって 15 年間、長いようで短いようで、やっぱり短かったなあというところでしょうか。その分、中身はほんとうに濃いものでした。このような貴重な経験ができたのも、皆さんからの励ましと協力が支えてくれたからであると、心から感謝しています。

ありがとうございました。まだまだ、学び続けたいと思います。もっともっと知らないこと、知るべきことを探求していきたいと思います。それじゃ！