情報通信工学実験IB

岡山県立大学情報工学部情報通信工学科
国島丈生（kunishi@c.oka-pu.ac.jp）

2012-09-24　第1.0版
2014-10-07　第1.1版（一部改訂）

予備知識

凡例

本資料では、以下の記法を用いる。

- タイプライタ書体(`irb`, `puts "Hello, World"` など)…コマンド名、Rubyプログラム、プログラムの出力結果など、コンピュータに対する入力・出力全般を表す。特に、以下の記法を用いる。
  - 下線付きタイプライタ書体(`irb` など)…ユーザからの入力を表す。
  - タイプライタ書体のイタリック体(*`Hello, World`* など)…プログラムからの出力を表す。
  - `expr #=> value` … 式 `expr` を評価した結果が値 `value` であることを表す。
- 特に断りがない場合、コマンドの実行例では、コマンドプロンプトとして $ を用いる。

Rubyプログラムの書き方

Rubyでは、構文の許容範囲がCより大きく、色々な書き方ができることが多い。その中には、思いがけず誤ったプログラムになってしまうようなものもある。Rubyに慣れていない人は、下記のページを見て、推奨される書き方をすること。もちろん、Rubyをすでに自習していてよくわかっている人については、このページの書き方に従う必要はない。

1. 筑波大学知識情報・図書館学類プログラミング演習II・プログラミング演習版 Ruby コーディング規約 [更新日 2007.09.05], http://klis.tsukuba.ac.jp/klib/Subjects/ProgI/toolbox/kiyaku.html

プログラムを読みやすくするコツ

以下の2つのCプログラムは全く同じ動作をするが、どちらが理解しやすいだろうか。

```
/* program 1 */
int main(){
int i,sum;
for(i=0;i<10;i++){
sum+=i;
}
printf("%d\n",sum);
}
```

```
/* program 2 */
int main() {
  int i, sum;
  for (i = 0; i < 10; i++) {
    sum += i;
  }
  printf("%d\n", sum);
}
```

多くの人が後者のほうが理解しやすいと感じるのではないだろうか。このように、プログラムの字下げ(インデント)を整えたり、空白を適宜入れることで、プログラムの読みやすさは大きく改善することができ、入力間違いによるバグを軽減することができる。

- インデントを整えることを習慣づけよう。特にプログラムのコピー&ペーストをしたり、修正を繰り返したりしているとインデントがむちゃくちゃになりがちである。Emacsを使っている場合は、1行入力し終えるごとに [tab] キーを押すと、適宜インデントを整えてくれる。
- 演算子の前後、予約語の前後、中括弧の前後などには空白を入れるよう、習慣づけよう。

本実験のサポートページ

本実験のサポートページを下記に用意している。本資料の最新版、説明用スライド、関連リンクなどをまとめていく予定なので、参考にされたい。

http://tk.kunilab.org/ja/experiments_ib

第1週：基本的な構文

ひとめぐりしよう

今回の主な内容について、演習形式で一通り学びましょう。

演習1

適当なエディタを用いて、次のような内容のファイル hello.rb を作成しましょう。

```
#!/usr/bin/ruby
print("Hello, World\n")
```

- ポイント
  - Rubyプログラムのファイルを作るときには、～.rb という名前にする。
  - Rubyプログラムの1行目には `#!/usr/bin/ruby` と書く。これはシェバング（shebang）と言い、Rubyプログラムの解釈実行コマンドが `/usr/bin/ruby` であることを表している。
  - 本実験で紹介するプログラムの動かし方の場合、シェバングはなくても動作する。しかし、他の書き方では正しく書かなければいけない場合もあるので、書くように習慣づけておいたほうがよい。これ以降出てくるプログラム例ではシェバングは省略しているので、各自で書くこと。
  - 2行目以降がRubyのプログラムである。

演習2

演習1で作成した hello.rb について、次のようにしてプログラムを実行しましょう。どのようになりましたか。

```
% ruby hello.rb
```

- ポイント
  - Rubyプログラムを実行する1つめの方法：ruby ファイル
  - ファイルに書かれたRubyプログラムが、上から順に順次実行されます。
  - Cのようにコンパイルする必要はありません。

- `print()` :Rubyのメソッド。「メソッド」とは、Cの関数に相当するものである（相違点については追々説明する）。`print()` は、引数で指定されたデータを文字列に変換し、標準出力に出力するメソッドである。
- `"Hello, World\n"` :Rubyの文字列。文字列は2重引用符で囲む（1重引用符で囲む場合もあるが、その違いは後述する）。最後の `\n` は改行を表す特別な文字。

演習3

irbというコマンドの使い方を学びましょう。次のように irb コマンドを実行しましょう。

```
% irb
irb(main):001:0> 1 + 1
=> 2
irb(main):002:0> print("Hello, World\n")
Hello, World
=> nil
irb(main):003:0> load("hello.rb")
Hello, World
=> true
irb(main):004:0> quit
%
```

- ポイント
  - irb はRubyプログラムを対話的に実行できるコマンドです。
  - irbを起動すると独自のプロンプト（以下の例では irb(main):001:0> など）が表示されます。そこにRubyプログラムを入力してリターンキーを叩くと、そのプログラムの評価結果が表示されます（上の例の => のところ）。
  - 入出力を用いるプログラムなどの場合には、評価結果以外にも表示されることがあります。
  - 作成済みのRubyファイルを読み込むにはload()という命令（メソッド）を使います。loadメソッドはファイルのロードに成功するとtrueを返します。
  - irbを終了するには、quitと入力してリターンキーを叩きます。
  - 左右のカーソルキーでカーソルの移動が行えます。
  - 上下のカーソルキーでコマンドの実行履歴がたどれます。

演習4

irbで以下の値を評価しましょう。評価結果はどうなりましたか。

1. 1
2. -2
3. 1.0
4. “hello, world”
5. ‘hello, world’
6. true
7. nil
8. false
9. [1, 3, 2, “aaa”]
10. []

- ポイント
    - これらはすべてRubyであらかじめ用意された定数の記法です。これをリテラルと言います。ここに挙げたのは代表的なリテラルです。
    - 1, 2：整数リテラル
    - 3：浮動小数点リテラル
    - 4, 5：文字列リテラル
    - 6, 8：真偽値
    - 7：「値がない」ことを表す特別なリテラル
    - 9, 10：配列
    - リテラルにも評価値があります。評価結果はそのリテラルそのものです。5だけ、1重引用符が2重引用符になるかもしれません。

演習5

irbで次の式を評価しましょう。評価結果はどのようになりましたか。

1. 1 + 2 * 3
2. 100 + 25.0
3. 1 == 1
4. 1 == 2

5. 1 == 1 && 2 < 3
6. 1 == 2 || 3 < 2
7. “aaa” + “bbb”
8. “aaa” != “bbb”

- ポイント
    - これらはRubyの演算子を用いた式です。
    - 1, 2：算術式。Cと似ています。
    - 3, 4, 5, 6：論理式。論理式の評価値は真偽値（true, false）になることに注意。
    - 7：文字列の連接。この + のように、同じ記号でも、演算数の種類によって違う意味になることがあります。
    - 8：文字列の比較も普通の比較演算子で行えます。

演習6

irbで次の式を順に評価しましょう。評価結果はどのようになりましたか。

1. x
2. x = 1
3. x
4. x += 3
5. x
6. x = “aaa”
7. x

- ポイント
    - 変数の宣言は必要ありません。最初にその変数に値を代入したときに自動的に宣言されます。
    - Cと同じような代入演算子が用意されています。
    - 1：代入文を実行する前なので、xという変数はまだ宣言されていません。ですので、エラーになります。
    - 代入文にも評価値があります。代入した値が評価値になります。
    - Rubyの変数には型がありません。ですので、2, 6のように、違う型の値を同じ変数に代入してもエラーにはなりません。

演習7

次のプログラムをenshuu7.rbというファイルに保存し、実行してみなさい。変数yの値を10や0にしてみて実行してみなさい。またこのプログラムを irb で実行し、評価値を調べなさい。

```
x = 10
y = 20
if x < y
  print(“y is larger than x”)
elsif x == y
  print(“y equals to x”)
else
  print(“y is smaller than x”)
end
```

- ポイント
  - Cと似た条件分岐が使える。Cに比べると、Rubyでは、予約語（else, elsifなど）、条件や分岐の記述（カッコや中カッコが不要）、改行の位置（上記のように改行すること）、空白の入れ方（予約語と条件の間には必ず空白がいる）などに違いがある。
  - 条件分岐にも評価値がある。評価値は、実行された条件分岐の最後の文の評価値である。
  - この場合は、printメソッドの評価値がnilであるため、全体の評価値もnilになる。

演習8

次のプログラムをenshuu8.rbというファイルに保存し、実行してみなさい。

```
sum = 0
i = 1
while sum < 50
  sum += i
  i += 1
end
print(sum, “\n”)
```

- ポイント
    - while文もCとほぼ同じことが書けます。ifの場合と同様、Cの書き方との違いに注意しておくこと。
    - while文の評価値は常にnilです。
    - printメソッドの引数を複数指定すると、引数を順次文字列に変換して標準出力に出力します。(sumに整数が格納されていることに注意)

演習9

次のプログラムをenshuu9.rbというファイルに保存し、実行してみなさい。

```
sum = 0
(1..5).each do |i|
  sum += i
end
print(sum, "\n")
```

- ポイント
    - RubyにはCのfor文に相当する構文がありません。while文で代用するか、上の例のように each メソッドで代用するのが一般的です。
    - eachメソッドの詳細は回を改めて説明しますが、ここでは、上の例を「1から5までの各整数 i について、sum += i を順次実行する」と読む、ということであると理解しておいて下さい。

演習10

次のプログラムをenshuu10.rbというファイルに保存し、実行してみなさい。最後の行の50を様々な値に変えて実行してみなさい。

```
def sum_above(n)
  sum = 0
  i = 1
  while sum < n
    sum += i
    i += 1
  end
```

```
  return sum
end

print(sum_above(50), "\n")
```

- ポイント
  - Cの関数相当のメソッドの使い方について述べます。
  - 関数を定義するときは def 関数名(引数) 〜 end という構文を使います。改行位置に注意。
  - メソッド内で（代入によって自動的に）宣言した変数（上の例の sum や i）はローカル変数であり、そのメソッド内でのみ有効です。
  - ここまで説明した範囲内では、メソッド外の変数を参照することはできません。

Rubyプログラムの構造

Rubyプログラムは式の並びであり、メソッド（関数）を宣言しなくてもよい（もちろん、必要と思えば宣言すれば良い。メソッドの構文は回を改めて説明する）。Rubyプログラムを実行すると、書き並べた式が順に実行されていく。

リテラル

Rubyのプログラム中に書く具体的な値をリテラル（literal）と呼ぶ。いくつかの種類があるが、まず基本的なものから見ていく。

数値リテラル

- 整数リテラル… 1, -2, +1, 1000000000
  - Cの short, int, long に相当するビット長の違いは、システムが自動的に使い分ける。ユーザは気にしなくてよい。
  - 整数リテラルはIntegerクラスのオブジェクトである。（クラス、オブジェクトについては第5週で詳述する）
- 浮動小数点リテラル… 小数点を含む数値は自動的に浮動小数点と扱われる。
  - 0.5, 3.141592, -163.5, 1.0, 4.3e10 (指数表記)
  - 精度は固定。C の float, double に相当する使い分けはない

    - 特殊な浮動小数点数
        - Infinity (正の無限大, 1.0/0.0 の結果など)、-Infinity (負の無限大, -1.0/0.0 の結果など)、NaN (非数, 0.0/0.0 の結果など)
    - Floatクラスのオブジェクトである。

真偽値

- 偽…nil, false
- 真…nil, false 以外の全ての値（trueも含む）

文字列リテラル

- 二重引用符（"）もしくは一重引用符（'）で囲まれた文字列
    - "Hello", 'Good morning'
    - 二重引用符の場合は、文字列中にエスケープ文字（"\n"など）を使うことができる。その他拡張機能については第2章で述べる。
- ⚠ Cとは異なり、文字列の最後に '\0' が置かれることはない。
    - …というか、'\0' が置かれるのはCだけです。

文字リテラル

- ?1, ?a など
    - Ruby 1.9以降では文字列を表す。すなわち ?1 と "1" は同じである。

特別な定数

- nil :「値がない」ことを表す。偽を表す真偽値にも用いられる。

変数と代入

- Rubyの変数（ローカル変数）は先頭が英小文字かアンダースコア、2文字目以降はASCII記号を除く文字かアンダースコア
    - 例：i, x, x_1, this_is_a_variable, thisIsAVariable
- 代入演算子… =, +=, -=, *=, /=, %=, **=, etc.
- 代入文の例
    - x = 5 #=> 5

    - name = “Kendai Taro” #=> “Kendai Taro”
- 多重代入
    - x, y, z = 1, 2, 3 #=> [1, 2, 3]
        - x = 1; y = 2; z = 3と同じ意味

⚠注意点

- Rubyの変数には型がない。
- 変数宣言は不要。代入を行ったときに自動的に変数が宣言される
- 代入された値自体が式の評価値になる
- インクリメンタル演算子（++, --）はない。i += 1, i -= 1などで代用

制御構造

Cの制御構造と似たものが用意されている。Cとほぼ同じ構文なので、容易に理解できるだろう。

⚠注意点

- if式にthen、while式にdoを含める書き方もあるが、今回は紹介しない。
- 中括弧を使わない
- 改行の場所に意味がある。下に示す構文のように改行を入れること。
    - 改行位置を変える方法はあるが、説明が煩雑になるので、ここでは省略する
- RubyにはC言語のfor文に相当する構文がない。代わりにイテレータ（iterator）と呼ばれるRuby独特の構文を使う。イテレータについては、第2週で説明する。

条件文

C の if 〜 相当

```
if 条件式
  文
end
```

C の if 〜 else 〜 相当

```
if 条件式
```

```
  文1
else
  文2
end
```

C の if 〜 else if 〜 else 〜 相当

```
if 条件式1
  文1
elsif 条件式2
  文2
else
  文3
end
```

繰り返し文

Cのwhile文相当

```
while 条件
  文
end
```

C の do 〜 while文相当

```
begin
  文
end while 条件
```

ループからの脱出

- break：ループを中断し、ループ外に脱出（Cの break 相当）
- next：ループの残りを実行せず、次の繰り返しにジャンプ（Cの continue 相当）
- redo：現在の繰り返しを再実行

C の switch 文相当

```
case 条件
when 値1
  文1
when 値2
  文2
else
  文3
end
```

演算子

数値リテラルに対する演算子

算術演算子

+, -, *, /, % (剰余), ** (ベキ乗)

```
1 + 2.0          #=> 3.0
# 演算数の型が自動的に変換される。
# 整数に揃えたい場合は値の変換メソッド（後述）を用いなければならない。
```

比較演算子

- ==, !=, <, >, <=, >=：結果は真偽値（true または false）になる。
- <=> (宇宙船演算子)
    - 1 <=> 2 #=> -1
    - 2 <=> 2 #=> 0
    - 3 <=> 2 #=> 1

真偽値に対する演算子

- 論理演算… &&, and, ||, or, !, not （論理和、論理積、否定ともに2通りの書き方）
    - not, ! の結果は true, false のいずれか
    - &&, and, ||, or の結果は演算数のいずれか（true, falseとは限らない）

- 例：false || 20　　#=> 20

文字列に対する演算子

- 連接（+）："abc" + "abc"　　#=> "abcabc"
- 比較：==, !=, <, >, <=, >=, <=>、辞書順で比較

メソッド

Cの関数に相当するものを、Rubyではメソッドという。メソッドについては回を改めて説明するが、ここではCの関数とほぼ同じ使い方のできる関数的メソッドの使い方、よく使うメソッドについてまとめておこう。

関数的メソッドの定義

関数的メソッド定義の構文は次の通りである。

```
def メソッド名(引数リスト)
  メソッドの本体 # 複数の式を並べて書く
end
```

関数的メソッドの定義、および呼び出しの例

```
def sum(x, y)
  x + y
end
def diff x, y
  x - y
end
sum(1, 2)　　#=> 3
diff(2, 1)　　#=> 1
```

メソッド呼び出しのときにはそれらが順次実行され、最後に実行された式の値がメソッドの戻り値になる。上の例では、式 x + y の評価値がメソッド sum の戻り値になる。戻り値として配列やハッシュを返すこともちろん可能である。

メソッドの途中で本体の実行を終了したい場合など、明示的にメソッドの戻り値を指定したい場合は return 式を用いる。

よく使うメソッド

⚠注意点

書式がCの関数呼出しとは異なる。
メソッドを適用したいデータ.メソッド名(引数)
となる。

引数の指定方法について、次のような違いがある。

- 引数がない場合、()は省略し、メソッド名だけにできる。
    - 例：1.to_f と 1.to_f() は同じ意味
- 引数がある場合もカッコを省略できる場合がある（演算子の優先順位の関係で省略できないこともある）
    - 例：print(“Hello, World”) と print “Hello, World” は同じ意味

本実験では、下記ページにあるコーディング規約に沿って書くことを推奨する。

出力

► print(文字列)

- 引数で指定した文字列を出力する。
- STDOUT.print(“Hello, World”)# STDOUT(標準出力)に出力
- print(“Hello, World”)　　# STDOUTは省略可能
- STDERR.print(“Hello, World”)# STDERR(標準エラー出力)に出力

► puts(文字列)

- 引数で指定した文字列を、改行を付けて出力する。
- STDOUT.puts(“Hello, World”)
- puts(“Hello, World”)　　# STDOUTは省略可能
- STDERR.puts(“Hello, World”)

入力

詳細は第4章に譲ることとして、とりあえず次のものだけ紹介しておく。

► gets

- 1行読み込み、読み込んだ文字列を返す。文字列の末尾には必ず “\n”（改行文字）が付くことに注意。
- 読み込むデータがなくなると、nilを返す。
- STDIN.gets　　# STDIN(標準入力)から1行読み込む
- gets　　# STDINは省略可能

文字列の末尾から改行を取り除く

► 文字列.chomp

- 文字列の末尾から改行文字を1つ取り除いた文字列を返す。末尾が改行文字でなかった場合は何もしない。
- 元の文字列は変更されないことに注意。
- “abc\n”.chomp　　#=> “abc”
- “abc”.chomp　　#=> “abc”
- “abc\n\n”.chomp　　#=> “abc\n”

値の変換メソッド

Rubyでは、ほとんどの場合、プログラマが明示的に値を変換してあげなければならない。値の変換メソッドの代表的なものとして、次のようなものがある。

► 数.to_i

- 「数」を整数へ変換した値を返す。
- 1.0.to_i　　#=> 1
- "3".to_i#=> 3

► 数.to_f

- 「数」を浮動小数点へ変換した値を返す。
- 1.to_f　#=> 1.0

► データ.to_s

- 「データ」を文字列へ変換した結果を返す。
- 1.to_s　#=> "1"
- 3.14.to_s　　#=> "3.14"

繰り返し

Rubyには、様々なデータに対して繰り返しを行うためにイテレータ（iterator）という独特の機能が用意されている。最もよく使われるイテレータは文字列や配列に対するものなのだが、これらについては回を改めて説明する。ここでは、Cのfor文の代用として、整数や範囲に対するイテレータを紹介しよう。

例

3.times { |i| puts i }
この例のメソッド times がイテレータであり、後ろの { |i| puts i } という部分（ブロック）を引数としてメソッドtimesを呼び出している。timesは繰り返しを行うイテレータであり、この例ではブロックを3回実行することになる。

□

構文

イテレータの構文は少々見慣れない形をしている。以下の2つの構文が用意されている。

```
データ.メソッド do |変数|
  文
end
```

データ.メソッド { |変数| 文 }
do～end, { ～ } の部分が繰り返される。変数には繰り返しごとに毎回異なる値がセットされる。この変数は do～end や { ～ } の中で用いることができる。ここで示すイテレータでは、Cのfor文におけるループ変数の役割を果たすと考えて良い。

注意点

ループとは異なり、イテレータを途中で脱出することはできない。

整数に対するイテレータ

繰り返し実行：times, upto, downto

```
3.times { |i| print i, "\n" }        # iには0, 1, 2が順にセット
1.upto(4) { |i| print i, "\n" }      # iには1, …, 4が順にセット
```

5.downto(2) { |i| print i, “\n” } # iには5, …, 2が順にセット
n.upto(m) の場合は n < m、n.downto(m)の場合は n > m でなければならない。

範囲に対するイテレータ

2つの値m, n（m < n）に対して、m .. n は「m以上n以下の値すべての並び」を表す。こういう値のことを、Rubyでは「範囲」と呼ぶ。範囲に対するイテレータの代表格は each である。

```
# iには2, 3, 4, 5 が順にセット
(2 .. 5).each { |i| print i, “\n” }

# iには“a”, “b”, “c” が順にセット
(“a” .. “c”).each { |i| print i, “\n” }
```

課題

1. 1から50までの奇数の和を求め、結果を標準出力に出力するプログラムをRubyで作成しなさい。while文を使用するもの、timesイテレータを使用するもの、eachイテレータを使用するもの、3通りで作ってみなさい。
2. 整数numが素数であるかどうかを調べるメソッドprime?(num)をRubyで作成しなさい。while文を使用するもの、いずれかのイテレータを使用するもの、2通りで作ってみなさい。[1]
3. フィボナッチ数列 $a_1 = a_2 = 1$, $a_n = a_{n-1} + a_{n-2}$ $(n \geqq 3)$ の第n項を計算するメソッドfib_rec(n)。再帰を用いて作成して下さい。
4. フィボナッチ数列 $a_1 = a_2 = 1$, $a_n = a_{n-1} + a_{n-2}$ $(n \geqq 3)$ の第n項を計算するメソッドfib_norec(n)。再帰を用いないで作成して下さい。
5. 2つの整数の最大公約数を求めるアルゴリズムとして、ユークリッドの互除法というアルゴリズムが有名である。これについて調べ、アルゴリズムの詳細について説明しなさい。
6. 2つの整数n, mの最大公約数を求めるメソッドgcd(n, m)。ユークリッドの互除法を用いてプログラムを作って下さい。

[1] Rubyでは、メソッド名の末尾に?や!を付けることがあります。慣習として、末尾に?を付けるとtrueかfalseを返すことを、! を付けると、そのメソッドを実行すると何らかのデータが変更される（副作用がある）ことを表します。

第2週：文字列、正規表現

ひとめぐり

演習1

次の式をirbで評価してみましょう。評価結果はどのようになりましたか。5, 6については、標準出力への出力結果もどのようになるか見て下さい。

1. ‘programming language’
2. “programming language”
3. ‘aaa\nbbb\n’
4. “aaa\nbbb\n”
5. print ‘aaa\nbbb\n’
6. print “aaa\nbbb\n”
7. ‘1 + 2 equals to #{1 + 2}’
8. “1 + 2 equals to #(1 + 2}”

演習2

次の式を順にirbで評価してみましょう。評価結果はどのようになりましたか。

1. s = “abcdefghijklmn”
2. s[0]
3. s[2..4]
4. s[-1]
5. s[1, 3]
6. s[0] = “A”
7. s

演習3

次の式を順にirbで評価してみましょう。評価結果はどのようになりましたか。

1. “aaa” + “bbb”
2. “aaa”.length

3. “abcde”.reverse
4. “abc de”.split()

演習4

次の式を順にirbで評価してみましょう。評価結果はどのようになりましたか。

1. /bc/ =~ “abcde”
2. /bc/ =~ “bcde”
3. /^bc/ =~ “abcde”
4. /^bc/ =~ “bcde”
5. /abb?c/ =~ “abcde”
6. /abb?c/ =~ “bcde”
7. /a(bc?).?(e?)$/ =~ “abcde”
8. $1
9. $2
10. “aaa:bbb:ccc”.split(/:/)

文字列

Rubyでは、文字列を表す記法がいくつも用意されている。ここではそのうち、よく使われるものを紹介する。

一重引用符による文字列リテラル

- 'Programming Language'
- 'aaa\'bbb' (文字列中に ' を含む場合は \ を付ける)

二重引用符による文字列リテラル

二重引用符で文字列を囲むと、以下の2つの機能が文字列中で使えるようになる。

- 改行文字（”\n”）、タブ文字（”\t”）などの制御文字
    - "aaa\nbbb"
- 式展開

- 文字列中の #{...} で囲まれた部分をRubyの式として評価し、その結果を文字列内に展開する
- i = 2
- "iの値は #{i}"　　#=> "iの値は2"
- "iの10倍は#{i * 10}"　#=> "iの10倍は20"

文字列操作

添字演算子

文字列に添字演算子を適用すると、部分文字列にアクセスできる。以下の例のように、添字の指定方法がCに比べて高機能である。これらは配列と同様の記法であるが、単にプログラマの利便性を考慮して同じ記法（メソッド）が用意されているに過ぎず、文字列が文字の配列になっているわけではない。

```
str = "abcdefghijklmnopqrstuvwxyz"
str[0]　　#=> “a”
str[2]　　#=> “c”
str[-1]　　#=> “z”
str[-2]　　#=> “y”
str[0..3]　　#=> “abcd”, strの0番目～3番目の文字からなる文字列
str[1, 4]　　#=> “bcde”, strの1番目の文字から長さ4の部分文字列

str[0] = "A"　#=> "A"
str　　#=> "Abcdefghijklmnopqrstuvwxyz"
```

その他の文字列操作

Rubyで用意されている文字列操作のうち、代表的なものを以下に挙げる。正規表現に関係するメソッドは後述する。

► 文字列1 + 文字列2

- 文字列1と文字列2をつないだ文字列を新たに作り、それを返す。「文字列の連接」という。
- "str" + "ing"　　#=> "string"

► 文字列 * 整数

- 文字列を整数回繰り返した文字列を新たに作り、それを返す。

- "a" * 5 #=> "aaaaa"

► 文字列.chop

- 文字列の最後の1文字を取り除いた文字列を新たに作り、それを返す。
- “abcde”.chop #=> “abcd”

► 文字列.chomp

- 文字列の最後の1文字が改行文字なら、それを取り除いた文字列を新たに作り、返す。改行文字でなければ、元の文字列をそのまま返す。（既出）

► 文字列.downcase

- 文字列中のアルファベットをすべて小文字に置き換えた文字列を返す。
- “Hello, World”.downcase #=> “hello, world”

► 文字列.upcase

- 文字列中のアルファベットをすべて大文字に置き換えた文字列を返す。
- “Hello, World”.upcase #=> “HELLO, WORLD”

► 文字列.capitalize

- 文字列の先頭の文字を大文字にする。先頭がアルファベットでなければ何もしない。
- “hello, world”.capitalize #=> “Hello, world”

► 文字列.length

- 文字列の長さを返す。
- "abcde".length #=> 5

► 文字列.reverse

- 文字列を逆順に並び替えた文字列を返す。
- "abcde".reverse #=> "edcba"

▸ 文字列.each_char { |char| … }

- 文字列に対するイテレータ。文字列の各文字charに対してブロックを実行する
- "aaa".each_char { |char| print char, "\n" }

シンボル

文字列と似ているデータとして、シンボル（symbol）というものが用意されている。Rubyの識別子もしくは文字列の前に : をつけた値である。以下はシンボルの例である。

- :"programming language"
- :id
- :method_name

シンボルは文字列とは異なり、ここまで紹介してきた文字列操作やメソッド、正規表現は使えない。また、プログラム中で一度定義すると、そのプログラムの実行中は一切変更できない。このような性質のために、ハッシュのキーとしてしばしば用いられる。シンボルは第6章で登場する。

正規表現

正規表現（regular expression）とは文字列のパターンを指定する方法であり、これを使うと文字列処理が飛躍的に便利になる。そのため、現代のプログラミング言語には大抵正規表現を扱う機能が用意されている。

正規表現の例

Rubyにおける正規表現の例を次に示す。

/abc?de/

この正規表現は「abdeまたはabcde」を表している。？はメタキャラクタ[2]と呼ばれ、ここでは c が0回か1回出現することを表している。

正規表現の利用例

正規表現を利用したプログラムの例を示そう。次に示すのは、標準入力から文字列を1行ずつ読み込み、その文字列に abde または abcde というパターンが含まれていれば標準出力に出力する、というプログラムである。

```
#!/usr/bin/ruby

while line = gets
  if /abc?de/ =~ line
    print line
  end
end
```

⚠注意点

- while文の条件の記述の仕方に注意。getsによって1行読み込み、その結果をlineに代入しているが、line = gets の評価値は代入した値そのもの、つまりこの場合は読み込んだ行の文

[2] 情報科学の学問（形式言語理論）の用語で言うと、？は正規表現で用意されている演算子であると言える。

字列そのものになる。したがって、読み込むデータがなくなると、getsがnilを返し、line = getsの評価値も nil (つまり偽) になってループから脱出する。

- 正規表現 =~ 文字列：右辺の文字列が、左辺の正規表現にマッチする部分文字列を含むかどうか判定する。詳細は後述。

正規表現の詳細

最も簡単な正規表現は文字の並びである。例えば、正規表現 /abc/ は “abc” という文字列にマッチする。

さらにRubyでは、メタキャラクタを正規表現中で用いることで、複雑な文字列パターンを指定することが可能である。以下によく使うメタキャラクタを列挙する。

パターン	マッチする文字列
\w	英数字およびアンダースコア（_）1文字
\W	英数字およびアンダースコア以外の文字
\s	空白文字（スペース、改行、タブ）1つ
\S	空白文字以外
\d	0から9までのいずれか1文字
\D	0から9以外の1文字
^	文字列の先頭
$	文字列の末尾
.	任意の1文字
[abc]	a, b, cのいずれか1文字。メタキャラクタ []を使うと、その中に並べた文字のいずれか1文字にマッチする。

[a-z]	aからzまでのいずれか1文字。メタキャラクタ [] の中で - を使うと、文字の範囲を指定できる。
?	直前のパターンが0回または1回繰り返されることを表す
*	直前のパターンが0回以上繰り返されることを表す
+	直前のパターンが1回以上繰り返されることを表す
|	選択。左右のパターンのいずれかにマッチするものにマッチする
()	パターンのグループ化。後述するキャプチャ変数と併用することで、マッチした文字列の部分文字列を取り出すことができる

正規表現のオプション

正規表現は、後ろにオプションを幾つか付けることができる。例えば、/ab/ は文字列 “ab” のみにマッチするが、/ab/i としてオプション i を付けると、大文字・小文字の区別をせずにマッチが行われるため、”ab”, “Ab”, “aB”, “AB” にマッチするようになる。

文字列と正規表現の照合

正規表現と文字列を照合するには =~ 演算子を用いる。式 /regexp/ =~ str は、文字列 str の中に正規表現 regexp に照合できる部分文字列があるかを調べ、あればその部分文字列の先頭位置を、なければ nil を返す。

```
/^a$/ =~ "a"          #=> 0
/ab/ =~ "bbbabd"      #=> 3
/abc/ =~ "bbbabd"     #=> nil
```

キャプチャ変数

Rubyの正規表現では「キャプチャ変数」という強力な機能が用意されている。正規表現中でメタキャラクタ () を用いると、照合が成功したときに、() の中のパターンに対応する部分文字列が自動的に変数 $1, $2, $3, ... （キャプチャ変数）に保存される。下のプログラムでは、標準入力から読み込んだ行に abde または abcde が含まれるときに、その行全体、ab または abc（$1に保存された文字列）、de（$2に保存された文字列）を出力する。

```
#!/usr/bin/ruby

while line = gets
  if /(abc?)(de)/ =~ line
    print line, “\n”
    print $1, “\n”
    print $2, “\n”
  end
end
```

文字列の分割

▸ 文字列.split(正規表現)

- 文字列を正規表現にマッチする部分で分割し、結果を配列で返す。
- 例: “aaa:bbb:ccc:ddd”.split(/:/)　　#=> [“aaa”, “bbb”, “ccc”, “ddd”]
- 引数を省略すると空白文字で分割する。
    - 例: “abc de”.split()　　#=> [“abc”, “de”]
- 引数に空の正規表現を指定すると、1文字ごとに分割する。
    - 例: “abcde”.split(//)　　#=> [“a”, “b”, “c”, “d”, “e”]

正規表現を用いた文字列の置換

▸文字列.sub(正規表現, 置換文字列), 文字列.gsub(正規表現, 置換文字列)

- 文字列中で正規表現にマッチした部分を置換文字列に置き換える。
- subメソッドは最初にマッチした部分だけ置き換える。一方、gsubはマッチした部分をすべて置き換える。
- 例: “abcabcabc”.sub(/bc/, “def”)　　#=> “adefabcabc”

- 例: “abcabcabc”.gsub(/bc/, “def”)　　#=> “adefadefadef”

課題

1. 次のデータを表す正規表現を作りなさい。
    a. 郵便番号を表す文字列。前3桁の数字と後ろ4桁の数字の間は ‘-’ (ハイフン記号)で区切られているものとする。
    b. 携帯電話の電話番号を表す文字列。携帯電話の電話番号は、市外局番の部分が 070, 080, 090 のいずれかである。なお、ハイフンは入っていても入っていなくてもよいように正規表現を作成すること。
2. 1.で作成した正規表現が意図通りになっているか検証するプログラムを作成し、様々なデータを入力として検証してみなさい。
3. 本実験サポートページに置いてあるファイル sample2-2.csv は、2008年2月から6月までの岡山県内2ヶ所での気温を集計したデータです。1行が1つの観測データを表していて、次のように、データ項目がコンマで区切られた形式になっています[3]。

    月,日,時,観測点1,観測点2

    このデータを読み込み、2ヶ所それぞれについて、月ごとに最高気温、最低気温、平均気温を出力するプログラムを作成しなさい。ファイルからの入力方法はまだ本実験で取り上げていないので、ruby kadai2-3.rb < sample2-2.csv とリダイレクトを使ってファイルを標準入力から読み込むようにしてください。(先を予習して、その内容を使うのでも構いません)
4. 英語で書かれたファイルの行数、単語数、文字数を出力するプログラムを書きなさい。「単語」とは「1つ以上の空白で区切られた(空白以外の)文字の並び」としておきます。また「文字」には改行や空白文字も含まれます。サンプルデータがサポートページにおいてあります(sample2-1.txt)。このファイルの場合、行数が27、単語数が2255、文字数が11840です。

参考:配列(今回の課題に必要なところだけ先取り)

いくつかの値を順にまとめたデータ構造。

- 記法としては、要素をコンマで区切り、角括弧で囲む。
- 入れ子になっていてもよい。
- 要素の型は同一でなくてもよい。
- 要素数は可変長である

[3] このような形式をCSV形式(Comma Separarated Values)という。

- 例
    - [1, 2, 3]
    - [1, "str", [1, 2, 3]]
    - []（要素数0の配列を表す）

配列の添字

配列の要素のアクセスには添字を用いる。
⚠ 使い方はCと同じだが、Rubyの配列は可変長であるため、以下のように配列長を越えた添字への代入が可能である。

```
array = [1, 2, 3]        #=> [1, 2, 3]
array[0]                 #=> 1
array[2] = 5             #=> 5
array                    #=> [1, 2, 5]
array[3] = “1”           #=> “1”
array                    #=> [1, 2, 5, “1”]
array[5] = "a"           #=> "a"
array                    #=> [1, 2, 5, “1”, nil, "a"]
```

配列のメソッド

配列にはあらかじめ様々なメソッドが用意されている。以下にその一部を示す。

► 配列.length, 配列.size

- 配列の長さを得る
- ["a", "c", "b", "b"].length #=> 4

第3週：配列、イテレータ

ひとめぐり

演習1

次の式を順にirbで評価しなさい。評価結果はどうなりますか。

1. a = [1, 2, 3]
2. a[0]
3. a[3]
4. a[-2]
5. a[1..2]
6. a[0, 2]
7. a[0] = 0
8. a
9. a[5] = “a”
10. a
11. a[1..3] = [7, 8, 9]
12. a
13. a[3, 3] = [5, 6]
14. a
15. a[2, 0] = [10, 11]
16. a

演習2

次の式をirbで評価しなさい。評価結果や出力結果はどうなりますか。

1. [1, 2, 3].length
2. [4, 2, 9, 3].sort
3. [1, 2, 2, 3, 3].uniq
4. [4, 5, 6].include?(5)
5. [4, 5, 6].include?(10)

6. [1, 2, 3].each { |i| print i, “\n” }
7. [“10”, “8”, “2”].sort
8. [“10”, “8”, “2”].sort { |x, y| x.to_i <=> y.to_i }
9. [1, 2, 3].map { |i| i * 2 }
10. [1, -2, 3].select { |i| i > 0 }

演習3

次の式を順にirbで評価しなさい。評価結果はどうなりますか。

1. h = { “a” => 1, “b” => 2 }
2. h[“a”]
3. h[“c”]
4. h[“a”] = 4
5. h
6. h.key?(“a”)
7. h.key?(“c”)
8. h.keys
9. h.values
10. h.each { |key, value| print key, “:”, value, “\n” }

配列

いくつかの値を順にまとめたデータ構造。

- 記法としては、要素をコンマで区切り、角括弧で囲む。
- 入れ子になっていてもよい。
- 要素の型は同一でなくてもよい。
- 配列の大きさは必要に応じて大きくなる。つまり要素数は可変である。
- 例
    - [1, 2, 3]
    - [1, "str", [1, 2, 3]]
    - []（要素数0の配列を表す）

配列の添字

配列の要素のアクセスには添字を用いる。文字列の場合と同様、負の添字、範囲などを指定することができる。

```
array = [1, 2, 3, 4, 5]   #=> [1, 2, 3, 4, 5]
array[0]                  #=> 1
array[-2]                 #=> 4, 最後から2番目の要素
array[1..3]               #=> [2, 3, 4], 1番目から3番目の要素からなる配列
array[2, 2]               #=> [3, 4], 2番目から2個の要素からなる配列
```

次に添字を用いた代入の例を示す。Rubyの配列は可変長であるため、配列長を越えた添字への代入が可能である。

```
array = [1, 2, 3]         #=> [1, 2, 3]
array[2] = 5              #=> 5
array                     #=> [1, 2, 5]
array[3] = “1”            #=> “1”
array                     #=> [1, 2, 5, “1”]
array[5] = "a"            #=> "a"
array                     #=> [1, 2, 5, “1”, nil, "a"]
```

負の添字や範囲によって代入先を指定することも可能である。

```
array[-1] = “b”           #=> “b”
array                     #=> [1, 2, 5, “1”, nil, “b”]
array[1..2] = [“A”, “B”]  #=> [“A”, “B”]
array                     #=> [1, “A”, “B”, nil, “b”]
```

代入する配列の大きさが、代入先の範囲と一致していなくても構わない。

```
array[2, 2] = [3, 4, 5]   #=> [3, 4, 5]
array                     #=> [1, “A”, 3, 4, 5, “b”]
```

要素を配列の途中に追加する場合は、長さ0の範囲を置き換えるという、ややトリッキーな方法を用いる。

```
array[3, 0] = [6, 7]      #=> [6, 7]
array                     #=> [1, “A”, 3, 6, 7, 4, 5, “b”]
```

配列のメソッド

配列にはあらかじめ様々なメソッドが用意されている。非常に数が多いので、ここではその一部のみ示す。

► 値.to_a

- 値を配列に変換する。
- (1..3).to_a #=> [1, 2, 3]

► 配列.length, 配列.size

- 配列の長さを得る
- ["a", "c", "b", "b"].length #=> 4

► 配列1 + 配列2

- 配列1と配列2を連接した配列を返す
- ["a", "c"] + ["d", "e"] #=> ["a", "c", "d", "e"]

集合演算

► 配列1 | 配列2

- 配列1, 配列2を集合とみて、集合和を求める。
- [“a”, “c”, “b”] | [“d”, “c”] #=> [“a”, “c”, “b”, “d”]

► 配列1 & 配列2

- 配列1, 配列2を集合とみて、共通集合を求める。
- [“a”, “c”, “b”] & [“d”, “c”] #=> [“c”

► 配列1 - 配列2

- 配列1, 配列2を集合とみて、集合差を求める。
- [“a”, “c”, “b”] - [“d”, “c”] #=> [“a”, “b”]

スタック、キュー

配列の先頭や末尾に要素を追加したり、先頭や末尾の要素を取り出す操作は頻繁に用いる。

- 配列の先頭に対する操作
    - 先頭に要素を加える: 配列.unshift(要素)
    - 先頭から要素を取り除く: 配列.shift
    - 先頭の要素を参照する: 配列.first
- 配列の末尾に対する操作

- 末尾に要素を加える: 配列.push(要素)
- 末尾から要素を取り除く: 配列.pop
- 末尾の要素を参照する: 配列.last

実行例を次に示す。

```
array = ["a", "c", "b"]   #=> ["a", "c", "b"]
array.unshift("e")        #=> ["e", "a", "c", "b"]
array.push("f")           #=> ["e", "a", "c", "b", "f"]
array.shift               #=> "e"
array                     #=> ["a", "c", "b", "f"]
array.pop                 #=> "f"
array                     #=> ["a", "c", "b"]
array.first               #=> "a"
array.last                #=> "b"
```

その他

► 配列.include?(値)

- 引数で指定された値が配列の中に含まれるかどうかを true, false で返す
- ["a", "c", "b", "b"].include?(“d”) #=> false

► 配列.sort

- 配列をソートした結果を新たな配列として返す（元の配列は変更されない）
- ["a", "c", "b", "b"].sort #=> ["a", "b", "b", "c"]

► 配列.uniq

- 重複した要素を削除した配列を返す
- ["a", "c", "b", "b"].uniq #=> ["a", "c", "b"]

► 配列.reverse

- 配列を逆順に並び替えた配列を返す
- ["a", "c", "b", "b"].reverse #=> [“b”, “b”, “c”, “a”]

イテレータ

いずれも配列の要素に対する繰返しを行う。ブロックに渡される引数の違いに注意。

► 配列.each { |item| … }

- 配列の各要素 item にブロックを順次適用して計算を行う。
- [1, 2, 3].each { |item| print item, “\n” } # 1, 2, 3が順次表示される

► 配列.each_with_index { |elem, index| … }

- 配列の要素と添字に対する繰返し。要素が elem に、添字が index にそれぞれ渡され、ブロックが実行される。

```
[“a”, “b”, “c”].each_with_index { |elem, index|
  print index, “:”, elem, “\n”
}
```

ブロック付メソッド

ブロックを指定するメソッドはイテレータ以外にもいろいろある。配列について、ブロックを指定する有用なメソッドをまとめておこう。

► 配列.sort { |item1, item2| … }

```
b = [“10”, “9”, “8”]
b.sort                                                    #=> [“10”, “8”, “9”]
b.sort { |x, y| x.to_i <=> y.to_i }          #=> [“8”, “9”, “10”]
```

ブロックなしでsortを実行すると、要素の型にあらかじめ用意されている大小比較に基づいてソートされる。この例だと、要素が文字列なので、文字列の大小比較の順、つまり辞書順でソートされる。
これに対して、3行目は大小比較の方法をカスタマイズしてソートを行っている。sortメソッドに要素の大小比較の方法をブロックとして渡している。x.to_i (xを整数に変換した値) と y.to_i (yを整数に変換した値)を大小比較している。

► 配列.map { |item| … }

要素全てにブロックを適用した結果からなる配列を返す。

```
[1, 2, 3, 4].map { |i| i * 2 }       #=> [2, 4, 6, 8]
```

► 配列.select { |item| … }

ブロックの評価結果が真となる要素だけを残した配列を返す。

```
[1, 2, 3, 4, 5].select { |i| i % 2 == 1 }              #=> [1, 3, 5]
```

ハッシュ

整数以外の添字を許すように拡張した配列をハッシュ（または連想配列）という。例を示そう。

```
months = {
```

```
"Jan" => 1, "Feb" => 2, "Mar" => 3, "Apr" => 4,
"May" => 5, "Jun" => 6, "Jul" => 7, "Aug" => 8,
"Sep" => 9, "Oct" => 10, "Nov" => 11, "Dec" => 12,
}
months["Jan"]          #=> 1
months["Aug"]          #=> 8
months["Oct"] = 13     #=> 13 キー "Oct" に対応する値の更新
months["Oct"]          #=> 13
months["None"]                 #=> nil
```

このように、Rubyでは、ハッシュのリテラルは { 値1 => 値2, ... } という形式をしている。値1（矢印の左辺）をキーという。ハッシュはキーと値の対応付けを行うデータ構造と考えられ、構造体のようなデータ構造を簡単に表現することができる。

ハッシュの要素の読み書きは、キーを添字として、配列と同様に行う。存在しないキーを指定すると、nilが返る。

ハッシュのメソッド

- ハッシュ.keys
  - ハッシュのキーのみを集めた配列を返す。
  - {"aaa" => 1, "bbb" => 2}.keys　　#=> ["aaa", "bbb"]
- ハッシュ.values
  - ハッシュの値のみを集めた配列を返す。
  - {"aaa" => 1, "bbb" => 2}.values　　#=> [1, 2]
- ハッシュ.length, ハッシュ.size
  - ハッシュの大きさ（登録されているキーの数）を返す。
  - {"aaa" => 1, "bbb" => 2}.length　　#=> 2
- ハッシュ.key?(キー)
  - 引数で指定したキーがハッシュに登録されているかを true/false で返す。
  - {"aaa" => 1, "bbb" => 2}.key?("aaa")　　#=> true
  - {"aaa" => 1, "bbb" => 2}.key?("ccc")　　#=> false
- ハッシュ.delete(キー)
  - 引数で指定したキーをハッシュから削除する。

```
h = {"aaa" => 1, "bbb" => 2}
        h.delete("aaa")                          #=> 1
h                                                #=> {"bbb"=>2}
```

ハッシュのイテレータ

- ハッシュ.each { |key, value| … }
  - ハッシュに登録されているキーkeyと値valueのペアについてブロックを順次実行する
  - {"aaa" => 1, "bbb" => 2}.each { |key, value|
    print key, ":", value, "\n"
    }

課題

1. 次のプログラムは何を印字するものか、説明しなさい。
   ```
   sum = 0
   (1..1000).to_a.select { |i| i % 3 == 0 }.each { |i| sum += i }
   print sum, “\n”
   ```
2. 与えられた数以下の素数をすべて求めるアルゴリズムに「エラトステネスのふるい」というものがある。
   a. どのようなアルゴリズムか説明しなさい。
   b. 整数n以下のすべての素数からなる配列を返すメソッド primes(n) を実装し、動作を確かめなさい。
3. 本実験サポートページに置いてあるファイル sample2-1.txtを各自でダウンロードし、ファイル中に含まれる単語の出現頻度を単語ごとに集計して出力するプログラムを作成しなさい。その際、次の条件を満たすようにして下さい。
   a. 1行に単語とその出現頻度を、1個のタブで区切って出力すること。例はこのようになる。
      the　103
      this　16
      that　28
   b. 結果の並べ替えは行わなくて良い。
   c. 簡単のため、大文字と小文字は区別して集計することとする。例えば、this, This, THIS は全て異なる単語として集計する。

d. ピリオド、コロン、セミコロン、引用符、ハイフン、数字など、英単語以外の文字列は含めずに集計すること。例えば、this. という文字列の場合は、ピリオドを取り除いて this として集計すること。
e. 簡単のため、単語の途中にこれらの記号が出現するような場合も、記号を削除してから集計してよい。例えば didn't は didnt という単語として集計する。

発展課題

課題3の条件b, c, eを次のようにしたプログラムを作成しなさい。

1. 出現頻度の降順に結果の並べ替えを行う。
2. 大文字と小文字は区別しないで集計する。例えば、this, This, THIS は全てthisとして集計する。
3. 単語の途中に引用符が出現する場合は削除せずに集計する。例えばdidn'tはそのまま didn'tとして集計する。

第4週：入出力

ひとめぐり

演習1

第2週や第3週で用いた sample2-1.txt を（カレントディレクトリにコピーするなどして）用意して下さい。その後、同じディレクトリに次の Ruby プログラムを ex4-1.rb というファイル名で保存し、実行しなさい。

```
file = File.open(“sample2-1.txt”)
while line = file.gets
  print line
end
file.close
```

演習2

前の演習に引き続き、同じディレクトリに次の Ruby プログラムを ex4-2.rb というファイル名で保存し、実行しなさい。

```
File.open(“sample2-1.txt”) { |file|
  while line = file.gets
    print line
  end
}
```

演習3

次のプログラムを ex4-3.rb というファイルに保存し、実行してみなさい。

```
Dir.open(“/”) { |dir|
  while name = dir.read
    print name, “\n”
  end
}
```

演習4

次の式を irb で評価してみなさい。

1. File.exist?(“/usr/include/stdlib.h”)
2. File.file?(“/usr/include/stdlib.h”)
3. File.directory?(“/usr/include”)
4. File.size(“/usr/include/stdlib.h”)
5. Dir.pwd
6. Dir.glob(“*”)
7. Dir.glob(“*.rb”)

演習5

次のプログラムを ex4-5.rb というファイルに保存し、実行してみなさい。

```
require ‘find’
Find.find(“.”) { |path| print path, “\n” }
```

準備

今回の解説に入る前に、これまで述べていない事項のいくつかについて先に説明しておきます。

- グローバル変数：$ で始まる変数はグローバル変数であり、プログラム中のどこからでも参照できます。
- 定数：大文字で始まる変数は定数であり、いったん定義すると書換えができません。クラス（後述）と区別するため、通常はすべて大文字にします。
- オブジェクト：ここまで見てきたように、Rubyで登場する「値」はすべてメソッドを持っており、C で扱う値とはかなり違います。そこで、以降、Rubyで登場する「値」を「オブジェクト」と呼ぶことにします。
- クラス：オブジェクトの構造（どのような値を内包し、どのようなメソッドを持つか、など）の定義を「クラス」と呼びます。例えば文字列はすべて String クラスに属します。また配列は Array クラスに、ハッシュは Hash クラスに、整数は Integer クラスにそれぞれ属します。

オブジェクトとクラスについては、次週、さらに説明を加えることにします。

プログラムへの引数

- 文字列の配列 ARGV を通してアクセスできる
    - ARGV の宣言や初期化は Ruby 処理系で行っているので、プログラマは気にしなくてよい
- 例：argv_sample.rb

```
# argv_sample.rb
print ARGV.join(', ')

% ruby argv_sample.rb 1 aaa 333
1, aaa, 333
```

入出力

標準入出力、ファイルへの入出力、URIへの入出力など、いくつか種類があるが、どれも基本的なメソッドは共通である。

標準入力、標準出力、標準エラー出力

- グローバル変数 $stdin , または組み込み定数 STDIN: 標準入力
    - gets は $stdin から1行読み込む
- グローバル変数 $stdout, または組み込み定数 STDOUT : 標準出力
    - p, print, puts は $stdout に出力する
- グローバル変数 $stderr, または組み込み定数 STDERR : 標準エラー出力
    - 警告やエラーメッセージの表示、デバッグ情報の表示などに用いられる

ファイル

ファイルからの読み出しやファイルへの書き込みは、Fileクラスのオブジェクトを用いる。

▸ File.open(ファイル名, モード)

- ファイルを開き、新しいFileオブジェクトを取得する。
- 第1引数はファイル名を表す文字列。
- モードの指定方法
    - “r” : 読み取りのみ（デフォルト）

    - “w” : 書き込みのみ
    - “r+” : 読み取り・書き込み、指定したファイルが存在しなければエラー
    - “w+” : 読み取り・書き込み、指定したファイルが存在しなければ新たに作成
    - “a” : 追記のみ
    - “a+” : 読み取り・追記
- 「クラス名.メソッド」という形のメソッドはクラスメソッドと呼ばれるものである。詳細は次回に譲る。今回のところは “File.open” という名前の関数的メソッドであると思っておいてよい。

▸ close

- Fileオブジェクトのメソッド。ファイルを閉じる。

使用例を以下に示す。

```
f = File.open(“sample.txt”, “r”);
# sample.txtに対する処理
while line = f.gets
  print line
end
f.close
```

ブロック付きメソッドを用いたファイルの処理

▸ File.open(ファイル名, モード) { |f| … }

- ファイルを開いて新しくFileオブジェクトを取得し、そのFileオブジェクトに対してブロックを実行する。
- ブロックの実行が終わると、自動的にファイルを閉じる
    - ファイルの閉じ忘れがなくなるので、お薦め。
- ブロックの引数には、取得したファイルオブジェクトが渡される。

使用例を以下に示す。

```
File.open(“sample.txt”, “r”) { |f|
  while line = f.gets
    print line
  end
}
```

入出力に共通する代表的なメソッド

以下に示すメソッドは、標準入出力、ファイル、いずれにも使うことができる。

▸ io.gets

- 入出力オブジェクト io から1行読み込み、読み込んだ文字列を返す。
- 行の末尾の改行が含まれることに注意。
- 入力の終わりに達してからさらに読み込むと nil を返す。

▸ io.readlines

- 入出力オブジェクト io から全ての行を読み込み、各行を要素とする配列を返す。

▸ io.read

- 入出力オブジェクト io から全てのデータを読み込み、一つの文字列として返す。

▸ io.each { |line| … }

- 入出力オブジェクト io から行を順次読み込み、各行に対してブロックを実行する。
- ブロックの引数 line には、読み込んだ行が代入される。
- 例

```
File.open(“sample.txt”, “r”) { |f|
  f.each { |line|
    print line
  }
}
```

▸ io.lineno

- gets や each メソッドで行単位で読み込みを行っている場合に、それまでに何行読み込んだかを返す。

▸ io.getc

- 入出力オブジェクト io から1バイト読み込む。
- 入力の終わりに達してからさらに読み込むと nil を返す。

▸ io.each_byte { |byte| … }

- 入出力オブジェクト io から1バイトずつ読み込み、それぞれについてブロックを実行する。

▸ io.puts(str0, str1, …)

- 引数すべてについて、最後に改行を付け、順にオブジェクト io に出力する。
- 引数が文字列以外のオブジェクトなら、自動的に文字列に変換してから出力する。

▸ io.putc(ch)

- 文字コードchに対応する文字をオブジェクト io に出力する。
- chが文字列ならば、先頭の文字を出力する。

▸ io.print(str0, str1, …)

- 引数で指定した文字列を順にオブジェクト io に出力する。
- 引数が文字列以外のオブジェクトなら、自動的に文字列に変換してから出力する。

▸ io.printf(format, arg0, arg1, …)

- formatで指定した書式に合わせてarg0, arg1, … を整形し、オブジェクトioに出力する。
- Cのprintf()に相当する。書式の書き方もほぼ同じと考えて良い（short や long などはRubyにはないので、それらを表す書式 %h, %l などはない。この他にも若干の違いはある）。

ファイル操作、ディレクトリ操作

普段、演習室環境で ls, mv, rm など、たくさんのコマンドを使用していると思います。ところで、これらのコマンドがどのように実装されているのか、考えたことはあるでしょうか? OSにはあらかじめ、ファイルやディレクトリを操作するライブラリ関数がたくさん用意されています。ls, mv, rm などのコマンドは、このようなライブラリ関数を使って実装されています。
ここでは、Rubyを通してファイルやディレクトリを操作する方法を紹介します。

ファイルの操作

▸ File.rename(oldfile, newfile)

- oldfile, newfileはいずれも文字列。
- ファイル oldfile の名前を newfile に変更する。
- 例: File.rename(“before.txt”, “after.txt”)
- 例: File.rename(“sample.txt”, “dir/sample.txt”)　　# 既にあるディレクトリの下に移す

▸ File.delete(file), File.unlink(file)

- fileは文字列。
- ファイル file を削除する。

ディレクトリの操作

▸ Dir.pwd

- 現在どのディレクトリで作業しているかという情報（カレントディレクトリ）を返す。
- プログラム実行開始時はコマンド pwd の実行結果と同じ。

▸ Dir.chdir(dirname)

- dirnameは文字列。
- カレントディレクトリを dirname に変更する。

▸ Dir.mkdir(dirname)

- 新しいディレクトリ dirname を作成する。
- dirname は文字列。

▸ Dir.rmdir(dirname)

- ディレクトリ dirname を削除する。
- dirname は文字列。
- dirname の中身は空（カレントディレクトリと親ディレクトリだけがある状態）でなければならない。

▸ Dir.open(dirname)

- ディレクトリ dirname を開いて新しく Dir クラスのオブジェクトを生成し、それを返す。
- dirnameは文字列。
- 例: dir = Dir.open(“/home/kunishi”)

▸ close

- Dirクラスのオブジェクトのメソッド。ディレクトリを閉じる。
- dir.close

▸ Dir.open(dirname) { |dir| … }

- ディレクトリ dirname を開いて新しく Dir クラスのオブジェクトを生成し、それを dir に代入してブロックを実行する。ブロックの実行が終了すると自動的にディレクトリを閉じる。

▸ read

- Dirクラスのオブジェクトのメソッド。そのディレクトリに含まれているもの（ファイル、ディレクトリなど）の名前を文字列として順に返す。
- カレントディレクトリを表す “.”、親ディレクトリを表す “..” も返ってくることに注意。

▸ each { |name| … }

- Dirクラスのオブジェクトのメソッド。そのディレクトリに含まれているものの名前を順に name に代入しながらブロックを実行する。

プログラム例を示そう。いずれもディレクトリ /home/kunishi 中にあるファイルやディレクトリの一覧を出力するものである。

```
# open, close, readを使用
dir = Dir.open(“/home/kunishi”)
```

```
while name = dir.read
  print name, “\n”
end
dir.close

# open, close, each を使用
dir = Dir.open(“/home/kunishi”)
dir.each { |name|
  print name, “\n”
}
dir.close

# ブロック付open, each を使用
Dir.open(“/home/kunishi”) { |dir|
  dir.each { |name|
    print name, “\n”
  }
}
```

▸ Dir.glob(パターン)

- パターンに合致するファイル名・ディレクトリ名の一覧を配列で返す。
- ここでは、よく使いそうなパターンの例だけ示しておくことにする。
- 例：Dir.glob(“*”)
  - カレントディレクトリ中のファイル・ディレクトリすべて
- 例：Dir.glob(“*.c”)
  - カレントディレクトリ中のファイルのうち、.c が最後につくものすべて

ファイルの属性検査

▸ File.exist?(path)

- パス path が存在すれば true を返す。存在しなければ false を返す。

▸ File.file?(path)

- パス path がファイルならば true、ファイルでなければ false を返す。

▸ File.directory?(path)

- パス path がディレクトリなら true、ディレクトリでなければ false を返す。

▸ File.readable?(path)

- パス path が読み込み可能なら true、そうでなければ false を返す。

▸ File.writable?(path)

- パス path が書き込み可能なら true、そうでなければ false を返す。

▸ File.executable?(path)

- パス path が実行可能なら true、そうでなければ false を返す。

▸ File.size(path)

- パス path のサイズを返す。path がファイルなら、そのファイルのバイト数である。ディレクトリの場合は、そのディレクトリの管理情報（含まれるファイルの情報など）のバイト数である。

ファイル名の操作

▸ File.basename(path), File.basename(path, suffix)

- パス path (ディレクトリを含んでいてもよい)から一番後ろの “/” 以降の部分を返す。
- suffix が指定されると、さらに末尾の suffix が取り除かれる。
- 相対パスや絶対パスからファイル名を取り出す時に用いられる。
- 例：File.basename("/usr/include/stdio.h")　#=> "stdio.h"
- 例：File.basename("/usr/include/stdio.h", ".h")　#=> "stdio"

▸ File.dirname(path)

- パス path から、一番後ろの “/” までの部分を返す。
- 相対パスや絶対パスからディレクトリ名を取り出すときに用いられる。
- 例：File.dirname("/usr/include/stdio.h")　#=> "/usr/include"

▸ File.expand_path(path)

- パス path を絶対パスに変換した結果を文字列で返す。

ファイル関連のライブラリ

ここまで示した File クラスや Dir クラスは、Rubyに最初から組み込まれている機能であり、特になにもしなくても使うことができた。Rubyには、組込み以外にもさまざまなライブラリが用意されており、少し準備をすると、これらのライブラリを使うことができる。中にはファイル操作に便利なものもあるので、一部紹介しよう。

ライブラリの読み込み方

requireというメソッドを使う。例えば:
require 'find'
と実行すると、以降、Findライブラリをプログラム中で使うことができるようになる。

⚠注意点

requireは通常、rubyプログラムの先頭に置き、メソッドの中などには置かない。Cプログラムでの #include と同じ位置に置くと考えるとよい。

ファイル操作に有用なライブラリ

- Findライブラリ（require ‘find’）
    - 指定したディレクトリ以下に存在するディレクトリやファイルを再帰的に処理するためのライブラリ。
    - Find.find(dir) { |path| … }
        - ディレクトリ dir 以下のすべてのファイルのパスを順に path に代入し、ブロックを実行する。
- Tempfileライブラリ（require ‘tempfile’）
    - 一時的な作業ファイルを管理するためのライブラリ。
- FileUtilsライブラリ（require ‘fileutils’）
    - ファイルの複製、移動、削除など、ファイルの操作を行うためのライブラリ。Fileクラスのメソッドよりも高機能である。

課題

1. Unixのコマンドにheadとtailというものがある。これに似た動きをするメソッド head(行数, ファイル名), tail(行数, ファイル名) を作成しなさい。それぞれ、引数で指定したファイルから先頭の行数分だけ、または末尾の行数分だけ出力するというものです。まず head コマンド、tail コマンドについてマニュアル等で調べ、実際にコマンドを実行してみて動作を確かめること。
2. ファイル from を to という名前のファイルに複製するメソッド copy_file(from, to) を実装しなさい。from, toはいずれも文字列とします。FileUtils クラスは使わずに実装しなさい。

3. 指定したディレクトリ中にあるファイルのうち、.dvi, .log, .aux という名前で終わるファイルをすべて削除するメソッド rmdvi(path) を実装しなさい。path はディレクトリ名を表す文字列とします。（これらのファイルは、LaTeXで文書作成をしたときに生成される一時ファイルです）
4. 指定したディレクトリ path 以下に含まれるファイルとそのサイズ（バイト数）を一覧表示するメソッド traverse(path) を次の2通りで実装しなさい。ファイル名は絶対パスで表示すること。（注意：ディレクトリは表示しない）
    a. Findクラスを使わないバージョン。（注意：ディレクトリ中にディレクトリが含まれる場合があるので、再帰的な処理をしなければいけない。その際、無限の再帰が行われないように注意すること。特に、カレントディレクトリと親ディレクトリの扱いに注意すること。）
    b. Findクラスを使ったバージョン。

発展課題

普段演習室のLinux環境を使っていて、自分が繰り返し実行しているファイル操作（操作の系列でも良い）がないか、思い出してみて下さい。その操作を実行してくれるRubyプログラムを設計し、実装してください。下記の点に注意すること。

1. いきなり複雑な操作を実装しようとしても難しい。最初は単純なものを考えてみること。
2. プログラムにバグがあると、大事なファイル（プログラムやレポートのファイルなど）を壊したり削除してしまったりしかねない。必ず、必要なファイルをコピーするなどして、テストのためだけの環境（sandbox, 砂場という）を作ってから試すこと。
3. レポートには、どのような操作を実現するプログラムなのか、プログラムの使い方などの説明も行うこと。

第5週：クラスとオブジェクト

ひとめぐり

演習1

irbで次の文を順次実行しなさい。評価結果はどうなりますか。

1. x = “abc”
2. x.object_id
3. y = “abc”
4. y.object_id
5. “abc”.class
6. “abc”.instance_of?(Array)
7. “abc”.instance_of?(String)
8. String.new
9. Array.new
10. Array.new(5)
11. Array.new(5, 1)

演習2

次のプログラムを queue.rb というファイルに保存しなさい。

```
class Queue
  def initialize
    @array = []
  end
  def enqueue(d)
    @array.push d
    return d
  end
  def dequeue
```

```
    data = @array.shift
    return data
  end
end
```

次にirbで以下の文を順次実行しなさい。評価結果はどうなりましたか。

1. load ‘queue.rb’
2. queue = Queue.new
3. queue.enqueue(1)
4. queue.enqueue(2)
5. queue.dequeue
6. queue.dequeue
7. queue.array

演習3

演習2で作成した queue.rb を次のように修正しなさい。

```
class Queue
  def initialize
    @array = []
    @name = “”
  end
  def enqueue(d)
    @array.push d
    return d
  end
  def dequeue
    data = @array.shift
    return data
  end
  def array
    return @array
  end
  def name
```

```
    return @name
  end
  def name=(value)
    @name = value
    return @name
  end
end
```

その後、irbを立ち上げ直して、以下の文を順次実行しなさい。評価結果はどうなりますか。

1. load ‘queue.rb’
2. queue = Queue.new
3. queue.enqueue(1)
4. queue.enqueue(2)
5. queue.array
6. queue.dequeue
7. queue.dequeue
8. queue.array
9. queue.array = [1, 2]
10. queue.name
11. queue.name = “A Queue”
12. queue.name

演習4

queue.rb をさらに次のように修正し、演習3と同じ文をirbで順次実行しなさい。評価結果は同じになりましたか。

```
class Queue
  attr_reader :array
  attr_accessor :name
  def initialize
    @array = []
    @name = “”
  end
  def enqueue(d)
```

```
    @array.push d
    return d
  end
  def dequeue
    data = @array.shift
    return data
  end
end
```

演習5

queue.rb をさらに次のように修正しなさい。

```
class Queue
  attr_reader :array
  attr_accessor :name
  def initialize
    @array = []
    @name = “”
  end
  def enqueue(d)
    @array.push d
    return d
  end
  def dequeue
    data = @array.shift
    return data
  end
  def Queue.create(array)
    queue = Queue.new
    array.each { |data|
      queue.enqueue(data)
    }
    queue
```

```
  end
end
```

次にirbを立ち上げ直し、次の文を順次実行しなさい。評価結果はどのようになりましたか。

1. load ‘queue.rb’
2. queue = Queue.create([1, 2, 3, 4, 5])
3. queue.array
4. queue.dequeue
5. queue.array

演習6

queue.rb を次のように修正しなさい。

```
class Queue
  attr_reader :array
  attr_accessor :name
  @@foo = “foo”
  def initialize
    @array = []
    @name = “”
  end
  def enqueue(d)
    @array.push d
    return d
  end
  def dequeue
    data = @array.shift
    return data
  end
  def Queue.create(array)
    queue = Queue.new
    array.each { |data|
      queue.enqueue(data)
    }
```

```
    queue
  end
  def foo=(str)
    @@foo = str
  end
  def foo
    @@foo
  end
end
```

次にirbを立ち上げ直し、次の文を順次実行しなさい。評価結果はどうなりますか。

```
        load ‘queue.rb’
queue1 = Queue.new
        queue2 = Queue.new
        queue1.foo
        queue2.foo
        queue1.foo = “bar”
        queue1.foo
        queue2.foo
```

演習7

次のプログラムをファイル another_queue.rb に保存しなさい。

```
class AnotherQueue < Array
  def enqueue(d)
        push d
  end
  def dequeue
        shift
  end
end
```

次にirbで以下の文を順次実行しなさい。結果はどうなりますか。

```
load ‘another_queue.rb’
queue = AnotherQueue.new
```

```
queue.enqueue(1)
queue
queue.length
queue.dequeue
queue
```

メソッド

Rubyのメソッドについて、ここで整理をしておこう。

メソッド呼び出し

Rubyでのメソッド呼び出しには次のような形をしていた。

```
1.to_s
"string".split(//)
```

この例では、1や "string" といったデータに対して、to_s や split といったメソッドを適用している。メソッドを適用するデータのことをレシーバという。この例では、1や "string" がレシーバである。Rubyの場合、どのようなメソッド呼び出しでも必ずレシーバが存在する。

メソッド呼び出しは、通常、以下のような形式になる。

レシーバ.メソッド名(引数リスト)

クラスメソッド

次の例では、クラスFileをレシーバとして、メソッドopenを呼び出していることになる。このように、クラスをレシーバとするメソッドのことをクラスメソッドという。

```
File.open(“sample.txt”) { … }
```

関数的メソッド

次の例のように、一見レシーバを持たないメソッド呼び出し（関数的メソッド）もある。この場合はレシーバが省略されている。

```
print(“Hello, World”)
```

戻り値

メソッド呼び出しを行うと、必ず何らかの値が返ってくる。これをメソッドの戻り値（または返り値）という。1.to_s の戻り値は "1", "string".split(//) の戻り値は ["s", "t", "r", "i", "n", "g"] である。
メソッドの戻り値をレシーバにして、さらにメソッドを呼ぶことも可能である。この場合、. 演算子は左結合なので、左から順に計算が行われることになる。

```
13456.to_s.split(//)                #=> ["1", "3", "4", "5", "6"]
```

オブジェクトとクラス

Rubyで扱うデータはすべてオブジェクト（object）と呼ばれる。Rubyでは、ここまでの講義で登場した文字列、配列、ハッシュ、さらには整数や実数といったものまで、すべてオブジェクトである。また、オブジェクトの定義をクラス（class）という。

オブジェクトの特徴

オブジェクトとは一体どのような特徴を持つものなのだろうか。正確な定義が決められているわけではないが、一般に、次のような特徴を持つデータをオブジェクトと言う。Rubyのオブジェクトも以下の特徴を全て持つ。

- メッセージを受け取る：オブジェクトに対し、外部からメッセージを送ることができる。メッセージを送られると、オブジェクトは対応する何らかの処理を行う。Rubyでは、オブジェクトに対するメソッド呼び出しがメッセージに該当する。
- 内部状態を持っている：オブジェクトはいくつかの状態を取ることができる。メッセージを送られることにより、オブジェクトの内部状態は変化することがある。Rubyでは、オブジェクト内に変数を定義することで内部状態を実現している。（例）文字列オブジェクトは、長さを内部変数として保持している。文字列に文字を追加すると、長さを表す変数の値が変化する。
- 識別子を持つ：オブジェクトはそれぞれ固有の識別子（identifier）を持ち、これにより個々のオブジェクトを区別することができる。Cで動的割り付けを行う際、割り付けられた番地を陽に扱うことがあるが、これは識別子の最も素朴な例である。Rubyではプログラムから番地を直接扱うことはなく、Ruby実行系内で個々のオブジェクトに一意に番号を振って識別子に用いる。

▸ obj.object_id

- オブジェクトobjの識別子を得るメソッド。
- 例：同じ文字列であっても、識別子が異なることに注意してほしい。

```
x = "abc"
x.object_id                    #=> 181020
y = "abc"
y.object_id                    #=> 159630
y = x
y.object_id                    #=> 181020
```

▸ obj.class

- オブジェクトobjがどのクラスに属しているかを求めるメソッド。
- 例："abc".class #=> String

▸ obj.instance_of?(class)

- オブジェクトobjがクラスclassに属しているかを判定するメソッド。
- 例："abc".instance_of?(String)#=> true

オブジェクトを中心とするプログラミングスタイルをオブジェクト指向プログラミング（object-oriented programming）と呼ぶ。Rubyはオブジェクト指向プログラミングをしやすくするように設計されたプログラミング言語である。

オブジェクトと構造体

Cでの構造体の使い方を思い出してみよう。概ね次のような流れであった。

1. 構造体の宣言（構造体の名前、内部変数の構成など）
2. 領域の確保（構造体型の変数の宣言、mallocによる動的確保など）、初期化
3. 利用
4. 確保した領域の解放（変数の有効範囲に基づく自動解放、freeによる明示的な解放など）

Rubyのオブジェクトの使い方は、Cの構造体の使い方とよく似ている。

1. クラスの宣言（クラスの名前、内部変数の構成、メソッドの定義など）
2. インスタンス化：オブジェクト（領域）の確保（動的確保）、初期化
3. オブジェクトの利用
4. オブジェクトの解放（自動解放）

次に示すのは、文字列（Stringクラス）において、上記の手順とプログラムコードとの対応を示したものである。

```
# Stringクラスの宣言は、あらかじめRuby処理系で用意されている
# Stringオブジェクト "abcde" を動的確保し、オブジェクトへの参照を変数 str に代入
str = "abcde"

# lengthメソッドの定義はStringクラスの定義の中に書かれている
str.length

# 変数 str に「どこも参照しない」ことを表す特別なオブジェクト nil への参照を代入
# オブジェクト "abcde" はどの変数からも参照されなくなったので、自動的に解放される
str = nil
```

ただし、文字列はよく使われるため、プログラマの手間を減らすための工夫がいろいろ行われており、一般的な使い方とは異なる部分もある。以下、クラスの宣言、インスタンス化、オブジェクトの利用について、一般的な方法を述べる。

クラスの宣言

クラスの宣言は、あらかじめRuby処理系で用意されている場合と、プログラマが自分で行わなければならない場合とがある。Integer, String, Array, Hash, Regexp などはすべて前者であり、プログラマは宣言をすることなくオブジェクトを使うことができる。

クラスを新たに宣言するには、class式を用いる。

```
class クラス名
  # 内部変数の定義、メソッドの定義など
end
```

例えば、待ち行列を表すクラス Queue の定義は次のように書ける。

```
class Queue
```

```
  def initialize
    @array = []
  end
  def enqueue(d)
    @array.push d
    return d
  end
  def dequeue
    data = @array.shift
    return data
  end
end
```

@で始まる変数はオブジェクトの内部変数であり、Rubyでは「インスタンス変数」と呼ばれる。インスタンス変数は、同一クラスのメソッドからはグローバル変数のように扱える。したがって、上の例では、メソッド initialize, enqueue, dequeue で変数 @array を共有している。

メソッドの定義は、クラス外部でのメソッド定義と同じく、def～endという構文を用いる。この例では、Queueクラスのインスタンス変数2個（@array）、メソッド3つ（initialize, enqueue, dequeue）を定義している。このうち、initializeはオブジェクトの初期化を行う特別なメソッドであり、オブジェクトがメモリ上に確保されると、そのときに必ず initialize メソッドが呼び出される。

インスタンス化

宣言したクラスに関連づけられたオブジェクトをメモリ上に確保する作業を、インスタンス化という。クラスのインスタンス化には new メソッドを用いる。newメソッド内から initialize メソッドが呼び出される。initialize メソッド宣言に引数がある場合は、newメソッドの呼び出しに対応する引数を付けなければならない。

```
queue = Queue.new
```

オブジェクトの利用

何も制限をかけなければ、メソッドはオブジェクト外部から自由に呼び出すことができる。

```
        queue = Queue.new
```

```
queue.enqueue(1)        #=> 1
queue.dequeue(2)        #=> 2
queue.dequeue                   #=> 1
queue.dequeue                   #=> 2
```

インスタンス変数の有効範囲

インスタンス変数はオブジェクトの外部からは読み書きすることができない。

```
queue = Queue.new
print queue.array           # エラー
```

インスタンス変数の読み書きをオブジェクト外部から行うには、必ずメソッドを経由しなければならない。例えば、次の例では、@arrayの値を読み出すメソッドを Queue クラスに追加する。Rubyでは慣習的に、インスタンス変数の読み出しメソッドの名前はインスタンス変数と同じにする。

```
class Queue
  def initialize
        @array = []
  end
  def enqueue(d)
    @array.push d
    return d
  end
  def dequeue
        data = @array.shift
        data
  end
  def array
    return @array
  end
end
```

すると、次のようにして@lengthの値を外部から読み出すことができるようになる。

```
        queue = Queue.new
queue.array                 #=> []
queue.enqueue 5                     #=> 5
queue.array                 #=> [5]
```

もしインスタンス変数に代入を許したいのであれば、そのためのメソッドを用意する必要がある。
Rubyでは、インスタンス変数に代入するメソッドの名前は慣習的に “インスタンス変数名=” とする。

```
class Queue
  def initialize
    @array = []
    @name = “”
  end
  def enqueue(d)
    @array.push d
    return d
  end
  def dequeue
    data = @array.shift
    return data
  end
  def array
    return @array
  end
  def name
    return @name
  end
  def name=(value)
    @name = value
  end
end
```

```
queue.name = “A Queue”       #=> “A Queue”
```

なぜアクセス制限が付いているのか

将来、クラスの実装を変更する必要が出てきたときに、変更の影響を最小限にするためである。

アクセスメソッドの簡単な定義方法

インスタンス変数とアクセスメソッドは対にして定義することが多いので、まとめて簡単に定義する方法が用意されている。

```
class Queue
  attr_reader :array
  attr_accessor :name
  def initialize
    @array = []
    @name = “”
  end
  def enqueue(d)
    @array.push d
    return d
  end
  def dequeue
    data = @array.shift
    return data
  end
end
```

attr_reader、attr_accessor はいずれもインスタンス変数（の @ の後ろの部分）を引数に取る。
attr_reader はインスタンス変数と読み出しメソッドを、attr_accessor はインスタンス変数、読み出しメソッド、書き込みメソッドの3つを、それぞれ自動的に定義してくれる。

attr_reader, attr_accessor の引数は文字列ではなくシンボル（第3週参照）であることに注意すること。

メソッドに対するアクセス制限

メソッドにも、オブジェクト外部から呼び出しできるかどうか、制限をかけることができる。

- public: どこからでも呼び出すことができる。
- protected: そのクラスのインスタンスメソッド、またはそのサブクラス（後述）のインスタンスメソッドからしか呼び出せない
- private: そのオブジェクトのインスタンスメソッドからしか呼び出せない。この場合、レシーバは必ず省略しなければならない（つまり、常にselfをレシーバとして呼び出す）

本講義では、制限の指定方法については省略する。

クラスメソッド

クラスメソッドを宣言するときには、メソッド名の前にクラス名をつける。以下の例では、引数arrayを基に新たなQueueオブジェクトを生成するクラスメソッドQueue.createを宣言している。

```
class Queue
  attr_reader :array
  attr_accessor :name
  def initialize
    @array = []
    @name = “”
  end
  def enqueue(d)
    @array.push d
    return d
      end
  def dequeue
    data = @array.shift
    return data
  end
  def Queue.create(array)
    queue = Queue.new
    array.each { |data|
```

```
    queue.enqueue(data)
   }
   queue
  end
end
```

通常のメソッドとクラスメソッドには、レシーバが異なる以外にも大きな違いがある。クラスメソッドは、通常のメソッドとは異なり、そのクラスのオブジェクトとは独立して置かれる。つまりそのクラスのオブジェクトが一つも作られていなくても、呼び出すことができる。

```
queue = Queue.create [1, 2, 3]
queue.dequeue                    #=> 1
queue.dequeue                    #=> 2
queue.dequeue                    #=> 3
```

クラス変数

@@で始まる変数をクラス変数という。クラス変数は、そのクラスのすべてのオブジェクトで共有される変数であり、インスタンスメソッド、クラスメソッドのいずれからでも用いることができる。

例えば、クラスQueueにクラス変数 foo を追加すると次のようになる。

```
class Queue
  attr_reader :array
  attr_accessor :name
  @@foo = “foo”
  def initialize
    @array = []
    @name = “”
  end
  def enqueue(d)
    @array.push d
    return d
  end
  def dequeue
```

```
    data = @array.shift
    return data
  end
  def Queue.create(array)
    queue = Queue.new
    array.each { |data|
      queue.enqueue(data)
    }
    queue
  end
  def foo=(str)
    @@foo = str
  end
  def foo
    @@foo
  end
end

        queue1 = Queue.new
        queue2 = Queue.new
        queue1.foo                  #=> “foo”
        queue2.foo                  #=> “foo”
        queue1.foo = “bar”          #=> “bar”
        queue1.foo                  #=> “bar”
        queue2.foo                  #=> “bar”
```

継承によるクラスの宣言

もう一つ、別の方法でQueueクラスを定義してみる。Rubyにあらかじめ用意されているArrayクラスを拡張して定義する方法である。この例では、Arrayクラスに2つのメソッドenqueue, dequeueを追加している。

```
class AnotherQueue < Array
```

```
  def enqueue(d)
        push d
  end
  def dequeue
        shift
  end
end
```

AnotherQueueクラスのオブジェクトは、enqueue, dequeue だけでなく、Arrayクラスのメソッドも適用することができる。

```
queue = AnotherQueue.new
queue.enqueue(1)
queue.length                    #=> 1
queue.dequeue                           #=> 1
```

このように、すでに定義済みのクラスを元に、一部拡張して新しいクラスを定義することを、クラスの継承（inheritance）という。

- 用語
    - Queue1（拡張先のクラス）から見て、Array（拡張元のクラス）をスーパークラスという
    - Array（拡張元のクラス）から見て、Queue1（拡張先のクラス）をサブクラスという

▸ obj.is_a?(class)

- オブジェクトobjがクラスclassに属するか、継承関係をさかのぼって調べる。
- 例：queue.is_a?(Array)　　#=> true

スーパークラスのinitializeメソッドを呼び出す

AnotherQueueクラスにインスタンス変数 @name を付け加えることを考える。このとき、AnotherQueueクラスのオブジェクトの初期化は、(1) Array から引き継いできたインスタンス変数の初期化 (2) AnotherQueueクラス固有のインスタンス変数の初期化 という2つの処理を行わなければならない。

Rubyのコードは次のようになる。superという文が、スーパークラス（この場合はArray）の同名メソッド（この場合はinitialize）の呼び出しを表している。

```
class AnotherQueue < Array
  def initialize
    super                          # (1)
    @name = “”           # (2)
  end
  def enqueue(d)
          push d
  end
  def dequeue
          shift
  end
end
```

(2)の処理が必要ない場合（すなわちサブクラスに固有のインスタンス変数がない場合）は、initializeメソッドをサブクラスで定義する必要はない。この場合、(1)は自動的に行われる。

ブロック付きのメソッド

ブロックを引数に取るメソッドも、普通のメソッドと同じように定義できる。メソッド内から引数のブロックを呼び出すには yield 式を用いる。

```
def foo
  yield "bar"
end
foo {|item| puts item}              #=> barと出力
```

これは、ブロック {|item| puts item} を引数としてメソッド foo を呼び出した例である。メソッド foo の本体の yield 式に来たところで、引数として渡されたブロックを実行する。このとき、yield 式の引数（この場合だと "bar"）をブロックに引数として渡す。その結果、ブロックの仮引数 item に "bar" が束縛された状態で puts item が実行され、bar と出力される。

課題

1. 書籍1冊を表すクラス Book を作成しなさい。
    a. ファイル名は book.rb とすること。
    b. 以下のインスタンス変数を用意すること。
        - @title：本のタイトル（文字列）
        - @author：著者（文字列）
        - @publisher：出版社（文字列）
        - @year：出版年（整数）
        - @price：定価（整数）
        - @isbn_no：ISBNナンバー（整数）。ISBNナンバーとは、全世界の書籍に一意に付けられた13桁の番号で、日本語の書籍の場合、裏表紙に「978」で始まる番号として（通常バーコードの下に）印刷されている。
    c. これらのインスタンス変数すべてに対して、読み出し、書き込み両方のアクセスメソッドを用意すること。
    d. 加えて、次のメソッドも用意すること。
        - to_s: 書籍の情報を見やすく整形した文字列を返す。整形の書式は特に定めないので、各自で工夫すること。
2. 書籍リストを表すクラス BookList を作成しなさい。
    a. ファイル名は book_list.rb とすること。
    b. 以下のインスタンス変数を用意すること。
        - @books：書籍のリストを表す配列。配列の要素は Book クラスのオブジェクトである。
    c. 次のメソッドを用意すること。
        - all：引数なし。すべての書籍からなる配列を返す。
        - find_by_isbn_no(number)：ISBNナンバーがnumberである書籍オブジェクトを返す。ない場合はnilを返す。ISBNナンバーの性質上、返るのは書籍オブジェクト1つかnilである。
        - find_by_title(title)：タイトルがtitleである書籍オブジェクトからなる配列を返す。同じタイトルの本があるかもしれないので、結果は配列とする。

- add(book)：引数bookは書籍オブジェクト。bookを書籍リストに追加する。ただし、bookと同じISBNナンバーを持つ書籍オブジェクトが書籍リスト中にある場合は、追加をしない。
- remove_by_isbn_no(number)：ISBNナンバーがnumberである書籍オブジェクトを書籍リストから削除する。
- to_s: 書籍リストを見やすく整形した文字列を返す。Bookクラスのメソッド to_s を使用すること。

3. アマゾンなどのサイトから上記のインスタンス変数に対応するデータを調べ、BookクラスやBookListクラスが正しく動いているか検証しなさい。

第6週：応用：HTTP

ひとめぐり

演習1

Web上のデータの場所を表す記法であるURIの仕組み、RubyでのURIの扱い方についての演習です。
以下の式を順にirbで実行しなさい。評価値はどのようになりますか。

1. require ‘uri’
2. uri = URI.parse(“http://www.example.com/”)
3. uri.scheme
4. uri.host
5. uri.port
6. uri.path
7. uri = URI.parse(“http://www.example.com:8080/blog/2012/index.html”)
8. uri.scheme
9. uri.host
10. uri.port
11. uri.path

演習2

Webでデータを記述するときのもっとも基本的な記法であるHTMLについて、基本的な書き方を知る演習です。
次のテキストを index.html というファイルに保存し、ブラウザで開けてみましょう。

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset=”utf-8”>
    <title>test index page</title>
```

```
  </head>
  <body>
    <p>This is a test page.</p>
    <p><a href=”http://www.oka-pu.ac.jp/”>Link to OPU Homepage.</a></p>
  </body>
</html>
```

1. <title>と</title>で囲まれた部分はブラウザのどこに表示されていますか。
2. <body>と</body>で囲まれた部分はブラウザのどこに表示されていますか。
3. ブラウザ内にはリンクが張られていると思いますが、そのリンクをクリックすると、なんというURIのページに飛びますか。またリンクは index.html のどこに対応していますか。

演習3

Webのプロトコルであるhttpについて、概要を知る演習です。
まず、次のRubyプログラムを mywebserver.rb というファイルに保存して下さい。

```
#!/usr/bin/ruby

require 'webrick'

server = WEBrick::HTTPServer.new(
  :Port => 8080,
  :DocumentRoot => File.join(Dir.pwd, "public_html")
)

Signal.trap(:INT) { server.shutdown }
server.start
```

次に、mywebserver.rbを保存したディレクトリに、public_htmlというディレクトリを作ってください。
最後に、ruby mywebserver.rb と実行して下さい。これでWebサーバの準備が完了です。

1. ブラウザで http://localhost:8080/ にアクセスしてみて下さい。ブラウザには何が表示されますか。また mywebserver.rb を実行しているターミナルにはどのようなメッセージが表示されますか。
2. 演習2で作成した index.html を public_html ディレクトリの下に移し、ブラウザで http://localhost:8080/index.html にアクセスしてみて下さい。ブラウザには何が表示されますか。また mywebserver.rb を実行しているターミナルにはどのようなメッセージが表示されますか。

演習4

次のプログラムを tcpclient.rb というファイルに保存し、演習3の mywebserver.rb を実行させた状態で、tcpclient.rb を実行して下さい。

1. どのようなデータが表示されますか。
2. mywebserver.rb を実行しているターミナルにはどのようなメッセージが表示されますか。

```
#!/usr/bin/ruby

require 'socket'

sock = TCPSocket.open("localhost", 8080)

sock.write("GET /index.html HTTP/1.0\n")
sock.write("Host: localhost:8080\n")
sock.write("\n")

while line = sock.gets
  print line
end

sock.close
```

演習5

演習3で作成した mywebserver.rb を次のように修正し、実行して下さい。

```
#!/usr/bin/ruby

require 'webrick'

server = WEBrick::HTTPServer.new(
  :Port => 8080,
  :DocumentRoot => File.join(Dir.pwd, "public_html")
)
server.mount_proc("/echo") { |req, res|
  res.body = req.path
}

Signal.trap(:INT) { server.shutdown }
server.start
```

1. ブラウザで http://localhost:8080/echo/a/b/1 にアクセスしてみましょう。何が表示されますか。mywebserver.rb を実行しているターミナルにはどのようなメッセージが表示されますか。
2. 演習4で作成したtcpclient.rb を以下のように修正して実行してみましょう。何が表示されますか。

```
#!/usr/bin/ruby

require 'socket'

sock = TCPSocket.open("localhost", 8080)

sock.write("GET /echo/index.html HTTP/1.0\n")
sock.write("Host: localhost:8080\n")
sock.write("\n")

while line = sock.gets
  print line
end
```

```
sock.close
```

Web

- サーバ・クライアント型の分散システム
- 3つの重要な構成要素
    - URI（Universal Resource Identifier）：Webで扱われる資源（リソース）の場所を表す記法
        - Web上の資源 = Web上に存在する情報すべて、と考えて良い
    - HTTP（HyperText Transfer Protocol）：サーバ・クライアント間の通信プロトコル
    - HTML（HyperText Markup Language）：Web上の資源を記述するための最も基本的な記法

URI (Uniform Resource Identifier)

- Web上の資源の識別子（identifier）
    - Web上の資源のURIは、すべてWeb上で一意に定まる
    - 現状のWebでは、URL（Uniform Resource Locator）と同義であると考えて良い
- 例
    - http://www.example.jp/blog/entry/11
- HTTPに関するURIの場合、3つの構成要素からなる
    - URIスキーム：URIが利用するプロトコルなど。この例だと http
    - ホスト名：その資源を提供するコンピュータ名。インターネット上で一意に定まる名前（FQDN, Fully Qualified Domain Name）にする。この例だと www.example.jp
    - パス：ホスト内での資源の識別子。最も簡単な場合、資源（ファイル）への絶対パスになる。この例だと /blog/entry/11
- より複雑な例
    - http://www.example.jp:8000/search?q=www&encoding=utf8
    - ホスト名に、HTTPで利用するポート番号が指定されている
    - パスの ？ 以降：GETパラメータ（Webアプリケーションへ渡すパラメータ）

HTTP (HyperText Transfer Protocol)

- インターネットのネットワークプロトコル階層ネットワーク
    - インタフェース層：物理的なケーブルなど、OSI参照モデルのデータリンク層・物理層に対応
    - インターネット層：IP、OSI参照モデルのネットワーク層に対応
    - トランスポート層：TCP, UDP、OSI参照モデルのトランスポート層に対応
    - アプリケーション層：具体的なインターネットアプリケーションで使われるプロトコル、OSI参照モデルのセッション層、プレゼンテーション層、アプリケーション層に対応
- HTTP = アプリケーション層のプロトコルの一つ。トランスポート層にはTCPを想定
    - クライアントからのリクエストに対してサーバがレスポンスを返す
    - ポート番号のデフォルトは80

クライアント側の処理手順

1. リクエストメッセージの構築
2. リクエストメッセージの送信
3. （サーバからレスポンスが返るまで待機）
4. レスポンスメッセージの受信
5. レスポンスメッセージの解析
6. クライアント側でのデータ処理

サーバ側の処理手順

1. （クライアントからのリクエストを待機）
2. リクエストメッセージの受信
3. リクエストメッセージの解析
4. 適切なアプリケーションプログラムへ処理を委譲
5. アプリケーションから結果を取得
6. レスポンスメッセージの構築
7. レスポンスメッセージの送信

リクエストメッセージ、レスポンスメッセージ

- いずれも3つの部分から構成される

    - スタートライン：メッセージの1行目
        - リクエストライン：HTTPメソッド、リクエストURI、プロトコルバージョン
        - ステータスライン：プロトコルバージョン、ステータスコード、テキストフレーズ
    - ヘッダ：メッセージの付加情報
        - 名前:値　という形をしている
    - ボディ：メッセージ本体（ない場合もある）
- ヘッダとボディは空行1つで区切られる

リクエストメッセージの例

この例ではリクエストボディはない。リクエストヘッダの下に改行が必要であることに注意すること。

GET /index.html HTTP/1.0

Host: www.example.com:8000

レスポンスメッセージの例

HTTP/1.1 200 OK

Content-Type: text/html; charset=utf-8

<html><body>I'm WEBrick.</body></html>

HTTPメソッド

- URIで指定した資源に対して行いたい処理を指定
- 主なHTTPメソッド
    - GET: 指定したURIの情報を取得する
        - 例：GET http://blog.example.com/1 …URIで指定された記事の内容を取得する
    - POST: 指定したURIに対する子資源の作成など
        - 例：POST http://blog.example.com/ … ブログ http://blog.example.com/ に新しい投稿という子資源を作成する
    - PUT: 資源の更新
        - 例：PUT http://blog.example.com/1 … ブログの投稿 http://blog.example.com/1 の修正版を投稿する
    - DELETE: 資源の削除

        - 例：DELETE http://blog.example.com/1 …ブログの投稿 http://blog.example.com/1 を削除する
- HTMLフォームから発行できるHTTPメソッドはGET, POSTのみ
    - これ以外のメソッドを発行する場合は何らかのプログラムを通さなければならない
    - PUT, DELETEをPOSTで代用することもある

ステータスコード

- 3桁の数字
    - 1xx: 処理中…処理が継続していることを示す。
        - クライアントはそのままリクエストを継続するなどの処理を行う
    - 2xx: リクエスト成功…リクエストが成功したことを示す
        - 200 OK: リクエスト成功。GETに対してリクエストが成功した場合は、レスポンスボディにその資源の内容が入る。
    - 3xx: リダイレクト。他の資源への転送
        - クライアントはレスポンスヘッダ中のLocationヘッダを参照して、新しい資源に対するリクエストを行う
        - 301 Moved Permanently: リソースの恒久的な移動
    - 4xx: クライアントエラー…原因はクライアント側のリクエストにある
        - 404 Not Found: リソースの不在
    - 5xx: サーバエラー…原因はサーバ側にある
        - 500 Internal Server Error: サーバ側に何らかの異常が起こっている

リクエストメッセージ、レスポンスメッセージの代表的なヘッダ

- Date: Tue, 06 Jul 2010 03:21:05 GMT … メッセージを作成した日時
- Location: www.example.com … リクエストを送るサーバ名が www.example.com である
- Content-Type: text/html; charset=utf-8 … メッセージのボディの内容を表す。この例ではHTML形式のテキスト（text/html）で、文字エンコーディングがUTF-8であることを表している。
- Content-Length: 5538 … メッセージボディのバイト数が5538バイトである
- Authorization: Basic dXNlcjpwYXNzd29yZA== … 資源にパスワードによる認証が必要な場合、認証に必要なユーザ名とパスワードを送信する。この例では認証方式（Basic認証）とユーザ名、パスワード（dXNlcjpwYXNzd29yZA==の部分。ユーザ名とパスワードをBase64エンコーディングと呼ばれる方式で符号化している。この例では user:password という文字列をエンコードしている）

Webで扱われるデータ

サーバとクライアントでデータ形式について合意さえ取れていれば（すなわちサーバとクライアントが共に分かる形式のデータであれば）、どんなデータでも扱うことができる。現在よく使われるデータは次のようなものがある。

- テキスト形式
    - ブラウザで表示させるテキスト…HTML, XHTML
    - 主にブラウザで表示させず、プログラムで処理することを主目的とするテキスト…Atom, RSS, JSON
- 画像…Jpeg, GIF, PNGなど
- 音声…MP3 (MPEG Audio Layer-3), MPEG4-AACなど
- 動画…MPEG4, H.264, WebMなど

HTML（Hypertext Markup Language）

- Webページを記述するための書式（マークアップ言語）
    - マークアップ言語はプログラミング言語ではない
- プレインテキストに、タグと呼ばれる特別な記号を埋め込み、テキストの各部分の意味を明確にする
    - 例：<title>～</title>…ページのタイトル、 ～ …段落
    - 開始タグ（<html>, <title>, <head>など）、終了タグ（</html>, </title>, </head>など）
    - ほとんどの場合、開始タグと終了タグは対にして用いる
    - 開始タグと終了タグで囲まれた部分：要素
    - 要素は入れ子にすることができる（例：<html><head>～</head><body> ～ </body></html>）
- 他ページへのリンク：リンクを張る文字列
    - href="..."の部分を属性という。属性は要素の開始タグに付けることができる。hrefの部分を属性名、"..." の部分を属性値という。
- HTML文書
    - html要素のみからなる文書
    - html要素の中に他の要素が入れ子状に含まれる
    - HTML文書=html要素を根とする木

- 1行目に、文書がHTMLで書かれていることを表す目印（<!DOCTYPE html>, DOCTYPE宣言）を置く
  - 以前は文書型宣言と呼ばれ、記法がもっと複雑であった。最近はこれだけでよい。
- 一般的に、ページの外見（見た目）はHTML文書には記述されない
  - HTML文書をどのように表示するかはWebエージェント次第
  - ページの見た目を指定する方法：スタイルシート（CSSなど）…本講義では触れない

HTMLで書かれたテキストファイル（HTML文書）の例

```
<!DOCTYPE html>
<html>
  <head>
      <title>はじめてのHTML文書</title>
  </head>
  <body>
      <p>Hello world!</p>
      <a href="http://www.oka-pu.ac.jp">岡山県立大学トップページ</a>
  </body>
</html>
```

WEBrickライブラリによるHTTPサーバ

Webサーバのプログラム例

以下に示すのは、Rubyで書かれたWebサーバの例である。この例では、次のように動くようになっている。

- ポート番号：8080
- 公開されるディレクトリ：~/public_html （ホームディレクトリにあるpublic_htmlディレクトリ）

つまり、このRubyプログラムを動かした状態で、ブラウザ等で http://localhost:8080/index.html にアクセスすると、~/public_html/index.html がHTTPで返されることになる。

```
#!/usr/bin/ruby
```

```
# WEBrickライブラリを用いるための宣言
require 'webrick'

# WEBrick::HTTPServerクラスのオブジェクトを生成
# HTTPリクエストに対して、このオブジェクトが実際の処理を行う。
# ポート番号はデフォルトの80番ではなく、8080番を指定
# HTTP経由で公開するファイルを置くディレクトリを :DocumentRoot で指定
# 以下の例では、./public_html を DocumentRoot に指定している。
server = WEBrick::HTTPServer.new(
        :Port => 8080,
        :DocumentRoot => File.join(Dir.pwd, “public_html”)
)
# プログラムに対する割り込みの指定。Ctrl-Cを押すとWebサーバが終了するようにする
Signal.trap(:INT) { server.shutdown }
# Webサーバの起動
server.start
```

ここまで述べていなかった構文やメソッドがいくつか使われているので、補足をしておこう。

- :Port, :DocumentRoot, :INTのように、Rubyの識別子（や文字列）の前に : を付けると、プログラム中で一意に定まる記号（シンボル）を表す。(第2章参照)
- WEBrick::HTTPServer.new(:Port => 8080, :DocumentRoot => File.expand_path("~") + "/public_html")
    - WEBrick::HTTPServerクラスのコンストラクタ呼び出し。ハッシュを引数として渡している。
- File.join：Fileクラスのクラスメソッド。引数で指定されたいくつかの文字列を、OSの区切り文字（Linuxだと ‘/’）を間に挟みながら連結した文字列を返す。
- Signalクラス：UNIXのシグナル関連の操作をするためのクラス
    - Signal.trapメソッド：Signalクラスのクラスメソッド。UNIXプロセスに対する割り込み処理を指定する。上のプログラムでは、:INT（interruput。Ctrl-Cを押したときに発生するシグナル）が発生すると server オブジェクトの shutdown メソッドを実行することを表している。

Webサーバを動かす

1. 上記のプログラムをhttpserver1.rbというファイルに保存する。
2. 保存したディレクトリに public_html というディレクトリを作る（mkdir public_html）
3. ruby httpserver1.rbと実行

すると、次のようなメッセージが表示され、HTTPサーバが立ち上がった状態になる。以降、HTTP通信のログ（記録）はこの端末上に表示される。

```
% ruby httpserver1.rb
[2012-11-06 05:30:37] INFO  WEBrick 1.3.1
[2012-11-06 05:30:37] INFO  ruby 1.8.7 (2012-02-08) [universal-darwin12.0]
[2012-11-06 05:30:37] INFO  WEBrick::HTTPServer#start: pid=77612 port=8080
```

⚠ 以下のようなメッセージが表示された場合は、8080番ポートがすでに使われている。同じプログラムを2つ以上実行させていないか（以前に実行させたプログラムを終了し忘れていないか）チェックすること。

```
[2012-11-06 05:30:37] WARN  TCPServer Error: Address already in use - bind(2)
```

この状態で、ブラウザで http://localhost:8080/ にアクセスすると、ブラウザにはそのディレクトリの中身が表示される。またログには次のように表示されるだろう。GET / というHTTPリクエストに対し、ステータスコード200が返され、462バイトのデータが返されたことが読み取れる。

```
localhost - - [06/Nov/2012:05:33:50 JST] "GET / HTTP/1.1" 200 462
- -> /
[2012-11-06 05:33:50] ERROR `/favicon.ico' not found.
localhost - - [06/Nov/2012:05:33:50 JST] "GET /favicon.ico HTTP/1.1" 404 281
- -> /favicon.ico
```

http://localhost:8080/favicon.ico へのアクセスも同時に発生している。これはブラウザのURL欄の左端に表示されるアイコン（favicon）を取得しようとして、ブラウザが自動的に行っているHTTPリクエ

ストである。~/public_html/favicon.ico はないので、ステータスコード404、「ファイルがない」ということを表す281バイトのメッセージが返されている。

次に、~/public_html/index.html として、次のようなHTMLファイルを用意する。

```
<!DOCTYPE html>
<html>
  <head>
        <title>test</title>
  </head>
  <body>
        <p>This is a test.</p>
  </body>
</html>
```

そして、ブラウザで http://localhost:8080/index.html にアクセスしてみよう。このときのHTTP通信のログを以下に示す。GET /index.html HTTP/1.0 というリクエストメッセージに対し、ステータスコード200 を返したことが読み取れる。

```
localhost - - [06/Nov/2012:05:43:34 JST] "GET /index.html HTTP/1.1" 200 120
- -> /index.html
```

HTTPServer#mount_procによる機能追加

WEBrickには、任意のRubyプログラムを実行し、その結果をHTTPレスポンスとして返す機能が用意されている。WEBrick::HTTPServerクラスのオブジェクトのメソッド mount_proc を利用する。以下に示すのは、HTTPリクエストのURIのパスをそのままHTTPレスポンスとして返すプログラムである。

```
#!/usr/bin/env ruby
require 'webrick'

s = WEBrick::HTTPServer.new(
  :Port => 8000,
```

```
  :DocumentRoot => File.join(Dir::pwd, "public_html")
)
s.mount_proc("/echo") { |req, res|
 res.body = req.path
}

Signal.trap("INT"){ s.shutdown }
s.start
```

▸ mount_proc(path) { |req, res| … }

- pathに対するHTTPリクエストに対するHTTPレスポンスメッセージの組み立て方を定義するメソッド。
- ブロックに対する引数 req, resはそれぞれHTTPリクエストメッセージ、HTTPレスポンスメッセージに対応するオブジェクトである。
- この例では、res.body（HTTPレスポンスメッセージのボディ）に req.path（HTTPリクエストメッセージ中のURIのパス）をセットしている。したがって、たとえば http://localhost:8000/echo/a/b/1 にアクセスすると、/echo/a/b/1 と表示される。

HTTPクライアントの実現

次にHTTPクライアントの実現方法をいくつか示す。

URIの処理：URIクラス

URIを処理するには、URIクラスを用いると便利である。

▸ URI.parse(uri)

- 引数 uri は、URIを表す文字列。
- uriを解析（パース）し、URIオブジェクトを返す。

▸ uri.host, uri.port, uri.path

- uri：URIオブジェクト
- uriのホスト、ポート、パスを返す。

URIクラスの使用例を次に示す。

```
uri = URI.parse("http://www.example.com:8080/blog/2012/01/article.html")
uri.host                    #=> "www.example.com"
```

```
uri.port        #=> 8080
uri.path                #=> "/blog/2012/01/article.html"
```

open-uriライブラリを利用したクライアント

Rubyのopen-uriライブラリを使うと、URIから取得したデータ（HTTPレスポンスボディのデータ）をファイルと同じように扱うことができる。以下の例は http://localhost:8080/index.html の内容を表示するプログラムである。requireの指定に注意すること。

```
require ‘open-uri’
open(“http://localhost:8080/index.html”) { |uri|
  while line = uri.gets
    print line
  end
}
```

Net::HTTPライブラリを利用したクライアント

open-uriでは、GETメソッド以外のHTTP処理はできない。より細かくHTTPを制御するにはNet::HTTPライブラリを用いる。ただし、open-uriと異なり、行単位で取得するといったことはできない。以下は、同じく http://localhost:8080/index.html の内容を表示するプログラムである。

```
require ‘net/http’
require ‘uri’
Net::HTTP.get_print(URI.parse(“http://localhost:8080/index.html”))
```

TCPソケットを利用したクライアント

HTTPはTCPを用いた通信であるので、一般的なTCP通信を使っても実現することができる。open-uriやNet::HTTPはHTTPに特化したライブラリであり、ヘッダやボディの処理をある程度自動でやってくれるが、TCP通信を直接行う場合は、ヘッダやボディの処理を自前で行わなければならない。したがって、普通はopen-uriやNet::HTTPを使うほうがよいだろう。

ここでは、リクエストメッセージやレスポンスメッセージを可視化するために、あえてTCPソケットライブラリ(TCPSocketクラス)を用いたHTTPクライアントの例を示す。http://localhost:8080/index.html からのレスポンスメッセージを（ヘッダも含めて）表示するプログラムである。

```
#!/usr/bin/ruby
```

```
require 'socket'

sock = TCPSocket.open("localhost", 8080)

sock.write("GET /index.html HTTP/1.0\n")
sock.write("Host: localhost:8080\n")
sock.write("\n")

while line = sock.gets
  print line
end

sock.close
```

課題

1. 「open-uriライブラリを利用したクライアント」で示したクライアントプログラムは、http://localhost:8080/index.html にしか対応していない。これらを改良し、コマンドラインの引数でURIを指定できるようにしなさい。例えばこのように実行させる。
   % ruby myprogram.rb http://www.c.oka-pu.ac.jp/
2. 「Net::HTTPライブラリを利用したクライアント」で示したクライアントプログラムについて、1と同様の改良を行いなさい。
3. 「TCPソケットを利用したクライアント」で示したクライアントプログラムについて、1と同様の改良を行いなさい。ただし、この資料で示した以上に複雑なURIは考慮しなくて構いません。具体的には情報通信工学科のWeb（http://www.c.oka-pu.ac.jp/）で公開されているHTMLページが扱えれば十分とします。
4. 演習3の mywebserver.rb を修正し、http://localhost:8080/now にアクセスすると現在の時刻が表示されるようにしなさい。なお、現在の時刻は Time クラスのクラスメソッド now で取得できます。
5. (成績評価には含めません) 6週間通しての感想、意見など、なんでも結構ですので、書いておいてください。

発展課題

第5回で作成したBookクラス、BookListクラスを利用して、演習3の mywebserver.rb を修正し、http://localhost:8080/all にアクセスすると本のデータの一覧が表示されるようにしなさい。

- 使用する本のデータは、あらかじめ mywebserver.rb 内に埋め込んでおいてよい。つまり、mywebserver.rb 内でいくつかBookオブジェクトを作成しておき、これらが表示されるようにすればよい。
- 一覧表示の方法は指定しない。HTMLでは、箇条書き（LaTeXの itemize 環境に相当）や表組み（LaTeXのtable環境に相当）などが指定できるので、Web等で調べて、適当なものを用いればよい。

参考文献

1. 筑波大学知識情報・図書館学類プログラミング演習II・プログラミング演習版 Ruby コーディング規約 [更新日 2007.09.05], http://klis.tsukuba.ac.jp/klib/Subjects/ProgI/toolbox/kiyaku.html
2. オブジェクト指向スクリプト言語Rubyリファレンスマニュアル, http://doc.ruby-lang.org/ja/1.8.7/doc/index.html