各種変数/データ型/演算

名古屋大学 情報基盤センター
情報基盤ネットワーク研究部門
基盤ネットワーク研究グループ
嶋田 創

ポイント

- 変数の型は重要だから常に意識しよう
- リスト(配列)はどのプログラミング言語にもあるし、重要だから覚えよう
  - ただし、覚えるのは「個々の要素へのアクセス方法」だけでOK
  - それ以外は言語によって違う所が多すぎる
- ディクショナリ(連想配列)は使えると便利だし、Light Weight Languageならまず使えるから存在は覚えておくべき
- リストやディクショナリや文字列に関する処理は言語によって可能/不可能の差が大きいので、頭の片隅に残す程度でOK
  - プログラムを組んでいる時に「そういえば、あの処理を使えば便利では」と思い出す→「参考書やウェブで検索して確認」とできるように

基本的な変数の型(復習)

- 文字列型(string): a = "abc"
  - 変換: str(*文字列型に変更する変数*)
- 整数型(integer): a = 47
  - 変換: int(*整数型に変更する変数*)
- 浮動小数点型(floating point): a = 1.13
  - 小数点がある数
  - 変換: float(*浮動小数点型に変更する変数*)

- アルファベット、数字(2文字目以降)、_が変数として利用可
  - Pythonの記述で使われる単語(予約語)は使用不可
- 型の違う変数を混ぜて操作しようとエラーが出る点に注意
  - 例: a = "abc", b = 3として、a + bを実行すると怒られます

Pythonで使える基本的な演算子(復習)

- 加算: a + b
  - 文字列を加算すると連結される
- 減算: a - b
  - 変数に単にマイナスをつけて正負反転も可能
- 乗算: a * b
  - 文字列と整数の乗算で文字列の繰り返し
- 除算: a / b
- 除算(小数点以下切り捨て): a // b
  - 負の数の切り捨ては「小さい値になる側に切り捨て」という点に注意
- 剰余: a % b
- 累乗: a ** b

リスト(配列) (1/3)

- 「*リスト名*[*インデクス*]」の形で定義されるデータ型
  - インデクスは数値
  - インデクスに変数を利用してもOK
  - (他のプログラミング言語では)一般的に「配列」と呼ばれる
- リストの宣言(新規作成)と一括代入: *リスト名* = [値1, 値2, 値3, ...]
  - 例: *prime* = [2, 3, 5, 7, 11]
- 個々の要素に代入: 「*リスト名*[*インデクス*]」に対して代入
  - あらかじめリストを宣言しておかないとエラーが出る
  - インデクスは0から始まる点に注意
  - 例: *prime*[0] = 2
    prime[2] = 5

リスト(配列) (2/3)

- リストを全部同じ値で初期化する場合(要素5個、0で初期化)
  - *(乗算)を使う: a = [0] * 5
  - for文を使う: a = [0 for *i* in range(5)]
- 「print(*リスト名*)」とすると、リストの内容全部が表示される
- リストの個々の要素を順番に表示する時にはfor文がよく使われる
  - 例1:

    for *i* in *リスト名* :

        print(*i*)
  - 例2:

    for *i* in range(len(*リスト名*)) :

        print("リストの" + str(*i*) + "番目の要素は" + str(*リスト名*[*i*]))

リスト(配列) (3/3)

- リストの次元数を増やすこともできる
  - 2次元配列(要は行列)の例: *リスト名*[*インデクス1*][*インデクス2*]

    image[0][0], *image*[1][0], ...

    image[0][1], *image*[1][1], ...

    ..., ...,
  - 2次元リストの宣言と代入:

    image = [[4, 3, 7, 6], [0, 2, 6, 5], [3, 1, 0, 6], [4, 3, 2, 1]]
    - 初期化にはfor文を使うのが良い: *image* = [[0 for *i* in range(4)] for *j* in range(4)]
  - 3次元、4次元とどんどん増やすことができる
  - 本格的に数値演算を行なうならばNumpyライブラリ、科学技術演算をやるならばScipyライブラリを利用した方が良い
- リストのサイズを大きくしすぎるとPCのメモリ上限に引っかかる可能性あり(4GBメモリ = 10億個のint型変数)

リストの操作(1/2)

- 値を追加: *リスト名*.append(*追加する値*)
  - 例: *prime*.append(13)
  - 空のリストに対してappendすることも可能
- リストの連結: *リスト名1* + *リスト名2*
  - *リスト名1*の後ろに*リスト名2*の中身を連結
- 要素を削除: del(*リスト名[インデクス]*)
- リストの範囲を取り出し: *リスト名[開始インデクス:終了インデクス]*
  - 例: *prime*[2:4]
  - 「*リスト名[開始インデクス:終了インデクス:ステップ]*」として「*ステップ*」おきに値を取り出すことも可能
- インデクスに負の値を使うことで、「リストの末尾から何番目」の値を指定することが可能

リストの操作(2/2)

- 値の挿入: *リスト名*.insert(*インデクス*, *値*)
  - 「*リスト名*[*インデクス*]」の場所に「*値*」を挿入
  - 元々の「*リスト名*[*インデクス*]」より後ろの値は1つずつ後ろにずれる
- 値の削除:*リスト名*.remove(*値*)
  - リストを最初からたどり、最初に一致した「*値*」を削除
  - 削除した値より後ろの値が1つずつ前にずれてくる
- 値の検索:*リスト名*.index(*値*)
  - 最初に一致した「*値*」のインデクスを返す
- 値の出現数のカウント:*リスト名*.count(*値*)
  - リスト中の「*値*」の数を数えて返す
- リストの長さ: len(*リスト名*)
- リストの中の数値の総和: sum(*リスト名*)
  - ただし、リストの中身が文字列だった場合は当然エラーになる

タプル(固定値配列)

- 後から値を書き換えることができないリストと考えればOK
- 宣言と値の一括代入: *prime* = (2, 3, 5, 7, 11)
  - 大括弧(リストの宣言)の代わりに小括弧を用いるとタプル
- 要素の参照はリストと同じく「*タプル名*[*インデクス*]」の書式
- タプルとリストは相互変換可能
  - list(*タプル名*)
  - tupple(*リスト名*)

セット(集合)

- 値の重複を許さないリストと考えればOK
- まとめて値を代入: prime = {2, 3, 5, 7, 11}
  - 中括弧を使う
- 集合の差分(引き算)や集合中に値が存在するか確認も可能
  - 値の存在の確認:「*値* in セット名」→「True」/「False」の結果
    - 例: (上のprimeに対し) 2 in prime → True
    - 型が違っていると「存在しない」と判定される点に注意
      - 例: (上のprimeに対し) "2" in prime → False
- 和集合や共通部分などの集合に対する演算が可能
  - 和集合: prime1 | prime2
  - 共通部分: prime1 & prime2
- リストやタプルと相互変換も可能
  - set(「*リスト名*」もしくは「*タプル名*」)

ディクショナリ(連想配列) (1/3)

- 文字列をインデクスにできる配列と考えればOK

配列

インデクス	値
0	85
1	93
2	75
3	86

ディクショナリ

キー	値
araki	85
fuchigami	93
shimada	75
yanase	86

  - ディクショナリにおいてインデクスは「キー」と呼ばれる
- (他のプログラミング言語では)一般的に「連想配列」と呼ばれる
  - ハッシュと呼ぶプログラミング言語もある

ディクショナリ(連想配列) (2/3)

- 宣言と一括代入: *ディクショナリ名* = { *キー1* : *値1*, *キー2* : *値2*, ...}
  - 例: *grading* = { 'araki' : 85, 'fuchigami' : 93, 'shimada' : 75, ...}
  - キーは「'」と「"」のどちらでくくってもOK
  - 複数行に分けて宣言するのもOK(見やすいし、編集しやすい)

```
grading = {
    'araki' : 85,
    'fuchigami' : 93,
    }
```

行頭のスペースは無くてもかまわないが、あった方が分かりやすい

カンマの後に中括弧で綴じ目も問題ない

  - 参考: if文などの条件文も、「¥」を行末につけて改行することで複数行化できる

```
if (a < b) and ¥
   (c > d) and (e > f) :
```

ディクショナリ(連想配列) (3/3)

- 個別に代入: *ディクショナリ名{キー名} = 代入する値*
  - キーに変数を使う場合は文字列型であることに注意
    - 変数は'や"でくくらないように注意
  - 例: *dict*['*key1*'] = *val1*, *dict*[*variable*] = *val2*
  - リストとは異なり、最初に宣言しておかなくて良い(いつでもキーと値のペアを追加可能)
- 存在しないキーの値を参照するとエラーでプログラムが停止するので注意
  - 標準入力などからの入力値をキーとする場合、「キーがあること」を確認してから処理しないと「キー無し」エラーでプログラムが停止
  - 「キーが無い場合の値」を設定できる*ディクショナリ名*.get()を使って参照すると安全
    - *ディクショナリ名*.get(キー, キーが無い場合の値)

ディクショナリの操作(1/2)

- キーと値の組を削除: *ディクショナリ名*.pop(*キー*)
  - ディクショナリを全部消す: *ディクショナリ名*.clear()
- ディクショナリのキー数を知る: len(*ディクショナリ名*)
- キーのリストを作成: *ディクショナリ名*.keys()
  - ただし、キーは脈絡ない順番で出てくるので、必要に応じて昇順ソートなどを「sorted(*ディクショナリ名*.keys)」などの形で適用
  - for文との組み合わせで個々のキーをループ処理に投入可能

```
for each_key in dict.keys():
    print("キー: " + each_key + " , 値: " + dict[each_key])
```

- 値のリストを作成: *ディクショナリ名*.values()

ディクショナリの操作(2/2)

- あるキーがディクショナリ中に存在するかどうか?: *検索するキー* in *ディクショナリ名*
  - 例: '*key1*' in *dict*
  - 結果はTrue(真) / False(偽)で返ってくる
  - 「in」の代わりに「not in」を使って「存在しない」の形の確認も可能
- ディクショナリの結合: *ディクショナリ名1*.update(*ディクショナリ名2*)
- 参照時にキーが無かった時に自動的にキーに対して初期値を入れる: *ディクショナリ名*.setdefault(*キー*, *初期値*)

文字列型に対する操作(1/3)

- 文字列の一部を切り出す: *文字列変数*[*開始文字番号*:*終了文字番号*]
  - リストの一部を切り出すのと同じ書式
  - 例: *sentence* = "abcdef"に対し、*sentence*[1:3]は"bcd"となる
    - 0から数える点に注意(プログラミング言語一般における約束)
  - 「開始文字番号」や「終了文字番号」を省くと「先頭から終了文字番号」「開始文字番号から末尾」となる
    - 例: *sentence* = "abcdef"に対し、*sentence*[1:]は"bcdef"、*sentence*[:3]は"abcd"となる
  - 負の番号を使うと末尾から数えることになる
    - 例: *sentence* = "abcdef"に対し、*sentence*[-3:]は"def"
  - もちろん、1文字だけ切り出すこともできる
- 文字列が数字として評価できるか: *文字列変数*.isnumeric()
  - 数字として評価できるならTrue、それ以外ならばFalseが返る

文字列型に対する操作(2/3)

- 文字列部分割: *文字列変数*.split(*区切り文字*)
  - 結果はリストの形で返る
  - よく空白を区切り文字として、英語の文を単語に分割するのに利用
    - 空白は「" "」の形で表す
- 文字列結合: *連結文字*.join(*文字列変数のリスト*)
  - 例: *sentence* = ["This", "is", "a", "pen."]
    " ".join(*sentence*)
- 文字列検索: *文字列変数*.find(*検索語*[, *開始位置*[, *終了位置*]])
  - 文字の開始位置が返ってくる
  - 見つからない時は-1が返る
  - 例: *position* = *sentence*.find(" a ")

文字列型に対する操作(3/3)

- 文字列を小文字に: *文字列変数*.lower()
- 文字列を大文字に: *文字列変数*.upper()
- 置換: *文字列変数*.replace(*置換前*, *置換後*)
- 他の変数の埋め込み: *文字列変数*.format(*val1*, *val2*, ...)
  - 文字列変数中の{0}, {1}, ...の部分に*val1*, *val2*, ...が埋め込まれる
  - 例: "変数1: {0}, 変数2: {1}".format(*variable1*, *variable2*)
- *文字列変数*.strip([*文字集合*])
  - 先頭および末尾にある「*文字集合*」の文字を削除
  - 通常は文字集合を指定せず、文頭/文末の空白の削除に利用
    - データ処理で、名前などの後ろに空白を入れられて困ることは多々ある
    - 特殊文字である改行文字も削除される