プログラミング実習I クラス5 (井村担当)

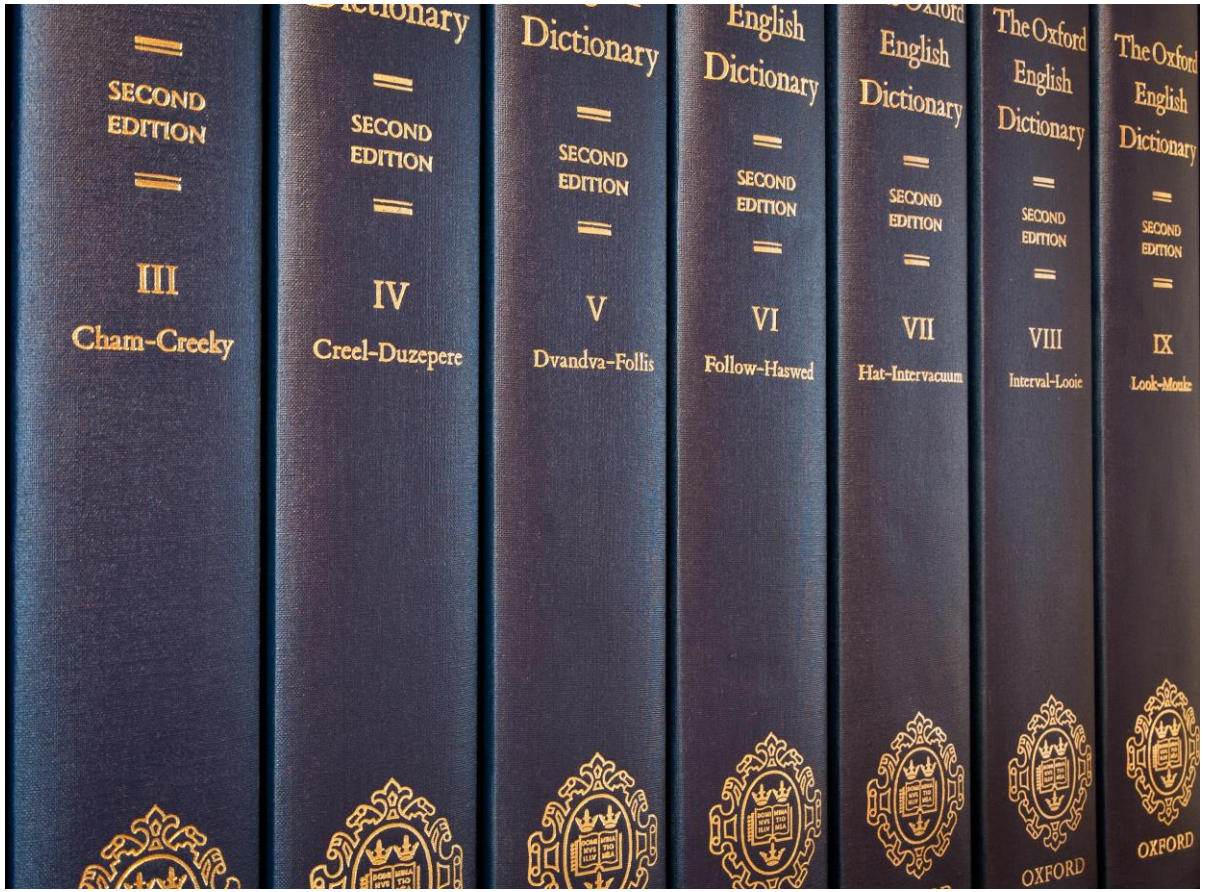
知能・機械工学課程 井村 誠孝

m.imura@kwansei.ac.jp

辞書型

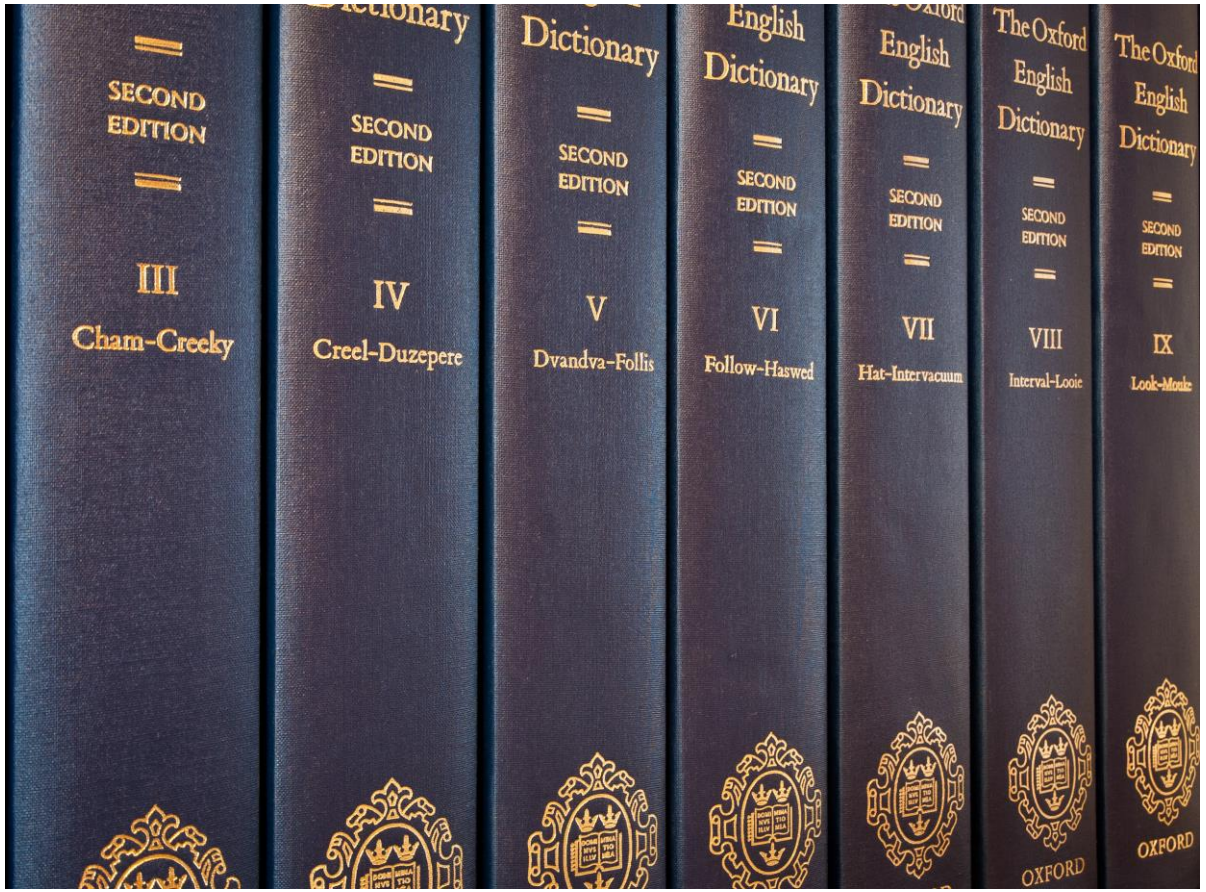
辞書のイメージ

- 見出しを調べると説明が書いてある.
- 何かの順に見出しが整列している.
- 何かのジャンルを網羅している.



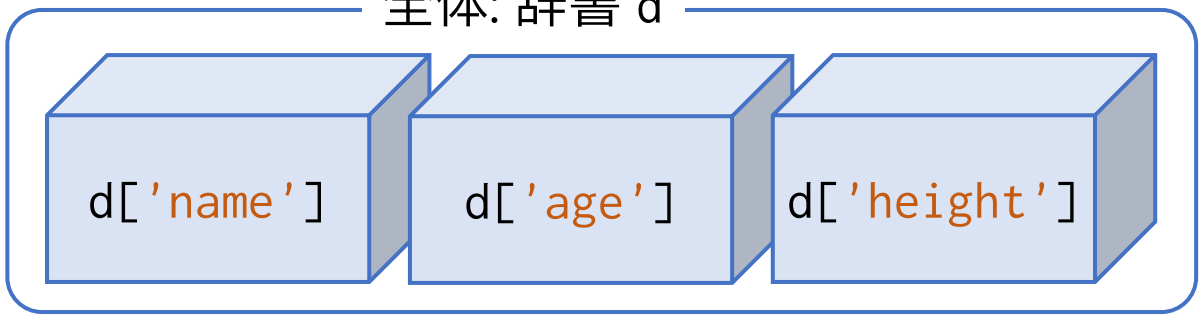
辞書のイメージ

- 見出しを調べると説明が書いてある．
  - Pythonの辞書でも，キーと値が対応付けられている
- ~~何かの順に見出しが整列している．~~
- ~~何かのジャンルを網羅している．~~



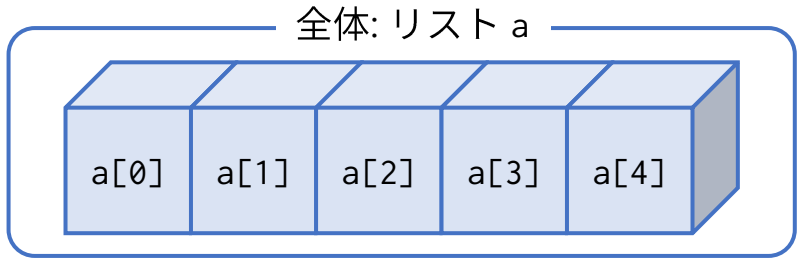
辞書型

- 辞書型 **dictionary** は，キー **key**と値 **value**を対応付けて記憶するデータ型である．
  - 他の言語では，連想記憶(associative memories)あるいは連想配列(associative arrays)と呼ばれることもある．
- キーを自由に設定可能．様々な型(整数，文字列など)を使える．
  - ただし，キーの内容が変化しない必要がある．リストをキーにすることはできない．後述するタプルはOK．



※この例ではキーは文字列ですが，整数などでもOK

参考: リストの場合



辞書型データの作成

- 波括弧({})の中に，キーと値を:で並べたペア(要素)を，カンマ(,)で区切って並べる．

```
dic = {1:'America', 39:'Italia', 86:'China'}
```

- キー，値とも，複数の型を混在させることが可能．

```
dic = {'name': 'Imura', 'age': 48, 'height': 170.2}
```

この例ではキーは単一の型です(文字列型)

- 要素の無い辞書(空辞書)

```
dic = {}
```

空の辞書を作ってどうする?
→後からデータを追加して使う

辞書の要素へのアクセス

- 辞書名の後ろに角括弧[ ]を付け，その中にキーを指定する(リストと同じ).
  - 作成するときに全体を囲むのは{}だが，参照するときは[]なので注意.

```
dic = {'name': 'Imura', 'age': 48, 'height': 170.2}
print(dic['height'])
```

```
170.2
```

  - キーが無ければエラーになる → キーの有無を調べる必要がある.

辞書にキーが存在するか調べる

- キーの有無を調べるには演算子 **in** を用いる．
  - in の返値は真偽型 (True or False)

```
dic = {'name': 'Imura', 'age': 48, 'height': 170.2}
print('name' in dic)
print('weight' in dic)
```

```
True
False
```

  - if 文などの制御構文と組み合わせて使う．

要素の追加と変更

- キーを角括弧[ ]内に記述し，値を直接代入する．
  - キーが辞書内に存在しない場合は，要素の追加となる．
  - キーが辞書内に存在する場合は，値の更新となる．

```
dic = {'name': 'Imura', 'age': 48, 'height': 170.2}
dic['weight'] = 59.0
dic['age'] = 49
```

更新された dic の内容

```
{'name': 'Imura', 'age': 49, 'height': 170.2, 'weight': 59.0}
```

  - 辞書内に同じキーは高々1個 (複数は存在しない)
  - 参考: 同じ値は複数存在してもよい

要素の削除

- メソッド pop() を用いる．
  - 指定されたキーを持つ要素(キーと値のペア)を削除し，削除された値を返す

```
dic = {'name': 'Imura', 'age': 49, 'height': 170.2}
dic.pop('age')
```

```
{'name': 'Imura', 'height': 170.2}
```

- del文を用いる．
  - この場合，値は返ってこない．

```
del dic['age']
```

辞書型とfor文

- for文のinの後ろに辞書型を記述すると，キーが順番に得られる．
  - キーに対応する値を得るには? → 得られたキーを使って 辞書名[キー]

```
country_code = {1:'America', 39:'Italia', 86:'China'}
for code in country_code:
    print(f'country_code[{code}] = {country_code[code]}')
```

```
country_code[1] = America
country_code[39] = Italia
country_code[86] = China
```

ややこしい? 一旦，キーのリストが得られると考えてもよい

- 辞書型に list() を適用すると，キーのリストが得られる．

```
country_code = {1:'America', 39:'Italia', 86:'China'}
list_code = list(country_code)
print(type(list_code))
print(list_code)
```

```
<class 'list'>
[1, 39, 86]
```

- よって以下は同じ

```
for code in country_code:
    :
```

```
list_code = list(country_code)
for code in list_code:
    :
```

参考: 辞書型からキーや値をまとめて得る

- キーの集合を得る: メソッド keys()
- 値の集合を得る: メソッド values()
- キーと値からなるタプル(後述)の集合を得る: メソッド items()
  - 返値は辞書ビューオブジェクトと呼ばれる型で，辞書の内容が変化すると，ビューオブジェクトの内容も変化する．
    - つまりメソッドが呼ばれた時点の状態をコピーして保持するわけではない．

```
country_code = {1:'America', 39:'Italia', 86:'China'}
for code, country in country_code.items():
    print(f'country_code[{code}] = {country}')
```

辞書型に関する落穂拾い

- 繰り返す際のキーの並び順は，ソートされてはいない．
  - キーとして異なる型を混在できるので，そもそもソートできない．
  - Python 3.7からは，キーの順序は辞書に追加された順序であることが保証されるようになった．
- 関数 len() を用いると，要素数が得られる．
  - リスト型や文字列型と同じ
- リスト型のように添字が連続していないので，スライスのように部分を取り出す機能はない．
- キーと値の区別をはっきりしておこう．両方とも自由度が高いので，どちらがどちらか混同しやすい．

タプル型

タプル型

- タプル型 tuple は，格納している要素が変更できないリスト型

タプルの作成

- 丸括弧()の中にカンマ(,)で区切って要素を並べる．

```
tuple_int = (0, 1, 2, 3)
```

実は丸括弧()は省略するとコードの解釈が曖昧になる場合を除いて省略可能だが，記述した方が人にわかりやすく，かつ省略できない場合は意外と多い．

- 複数の型を混在させることも可能．

```
tuple_mix = (2, 1.732, 'test')
```

- 要素が1個のタプル

```
tuple_one = (100,)
```

要素が1個の場合は，カンマが必要です．
カンマが無いと，整数と区別できません．
(実は，タプルを作るのはカンマなのです)

- 要素の無いタプル(空タプル)

```
tuple_empty = ()
```

リストと違い後からデータを追加できないので，使い道は限定的

タプルに対して可能な操作

- リストに対する操作のうち，内容を変更しないものは全て行える．
- 行えない操作の例
  - 値の更新
  - 要素の削除
  - 要素の追加
  - 並べ替え
- リストにできないがタプルにできること: 辞書のキーになる
  - 辞書のキーは不変である必要があるため