B演習 (言語処理系演習) 資料 (5)
データ表現

田浦

2006/12/4

1 概要

文字列の形で与えられたプログラムが，字句解析器によって字句の列に変形され，構文解析器によって構文木に変形された．構文木は，ようするにプログラムを，自然なデータ構造の形であらわしたもので，単なる文字列，あるいは字句の列よりも，処理系にとって扱いやすいプログラムの表現である．

そしていよいよ「評価器 (evaluator)」または「実行器 (executor)」と呼ばれる部品が，構文木として与えられたプログラムを「実行」する．評価器を作ることはインタプリタ構築の最終段階であって，これができれば，実際に mini-Python プログラムを実行することができるのである! ぜひともがんばってほしい．

さて，プログラムを「実行する」とはつまり，そのプログラムが起こすべき，外界へ見える副作用 (print 文，ファイル I/O の実行など) を忠実に実行することである．評価器が「何をしなければならないか」を分かりやすく説明する最も簡単な例は以下である．

```
print 1
```

という文を実行したら，画面 (正確にはインタプリタを実行しているプロセスの標準出力) に “1” と表示し，改行をしなくてはならない．

次の例として，

```
print x + 5
```

という式を実行するためには，まず x の値を求め，それに 5 を足した値を表示しなくてはいけない．念のために注意すると，この文は “x + 5” という文字列を表示するのではない．表示すべきなのはあくまで “x” という変数の中に入っている値 +5 であって，式そのものではない．

一般に，

```
print <式>
```

を実行したときに，表示されるべきなのはその<式>の「値」である．ということは評価器は当然のことながら任意の「式」の「値」を求める仕組みを内部に持っていなくてはならない．この値のこ

とを式を「評価した値」,「評価した結果」などという．つまりは評価器は内部に,「任意の式を与えられてその値を求める」仕組みを持っていなくてはならない．この仕組みのことを単に,「式を評価する」仕組みと呼ぶ.

評価器 (つまり評価器を実装している C のプログラム) の内部では，そのような「式を評価した値」は，ある約束事を守って C の値として表現される.「ある約束事」というのは，たとえば,「mini-Python の 10 は C の int 型の 10 で表現しますよ」などという約束である．当たり前に聞こえるかもしれないが，実際にはこれよりも複雑な約束が必要である．というのも，mini-Python の式を評価した結果は，整数とは限らないからである．評価器を作るためには，式を評価した値としてありうる値すべて，つまりは Python に現れうるすべての値について，それを評価器内部 (C のプログラム) でどう表現するかという約束が必要である．このような約束事を「データ表現」と読んでいる．

もちろん式だけでなく「文」の実行を行う必要もある．`print` 文自体が文のひとつであったから当たり前だが，もっと複雑な文 (for 文，while 文など) も実行できなくてはならない．それは，式の中には関数呼び出しがあり，関数呼び出しを評価するにはその関数の本体—それは文の並び—を実行する必要があるからである．たとえば,

```
print f(5)
```

を実行しようとすれば，`f(5)` という式を評価せねばならない．それにはまず，`f` という関数の本体を見つけ，そこに並んでいる文を実行していく必要がある．例えば `f(x)` の定義がどこかに,

```
def f(x):
    s = 0
    for i in x:
        s = s + i
    return s
```

などとかかれていたら，結局我々は代入文，for 文，return 文などを正しく実行できなくてはならない．こんな例を使って確認しなくても当たり前のことだが，要するにプログラムを意図通り実行するためには，mini-Python に現れるありとあらゆる式，文を，全体として整合性があうように評価・実行する仕組みを作らなくてはならないということである.

このようにして，評価器を正しく実装するということの輪郭が見えてきた．その輪郭とは,

- 式を評価した結果は，整数，浮動小数点数などの，mini-Python に登場する様々な値のどれかになる．これを「**Python 値**」と呼ぶことにしておこう.
- Python 値はインタプリタ内である約束事にしたがって C のデータとして表現される．まず我々は，**データ表現をしっかりと決めなくてはならない**.
- 式を評価する関数群が存在する．その役目は式の構文木 (`expr_t`) が与えられたら，その式があらわす Python 値をデータ表現の規則に従って作り出すことである.
- 同様に，文や，その並びを評価する関数群が存在する．その役目は式や式の並びの構文木 (`stmt_t` や `stmt_vec_t`) が与えられたら，それに応じた動作 (変数の値を書き換えたり，式の値を出力したりする) を行うことである.

2　目標の細分化

以上の議論を元に，評価器を作るという目標の細分化を行おう．

データ表現の決定: あらゆる可能な Python 値を今一度整理・列挙し，それらに対応する C プログラム内での表現 (データ表現) を決定する．

式の評価規則の擬似コードによる記述: 評価器のプログラム作成に入る前に，そもそも評価規則を整理し，全体の見通しを得るために，まず評価規則を擬似コード (自然言語を交えた，プログラム風の記述) で記述する．

文の評価規則の擬似コードによる記述: 同様のことを文に対して行う．

評価器プログラム作成: 擬似コードによる記述を参考にしながら，文，式，そして mini-Python プログラム全体の評価器を作成していく．

今週はこのうちの，データ表現について説明する．

3　データ表現の決定

3.1　mini-Python に登場するすべてのデータ型

mini-Python プログラムの実行中に登場する値 (つまり，変数に代入することができる値と考えればよい) には以下のものがある．これは大体，以前に Python 自身を紹介したときに明らかにしたことであるが，以下で若干注意を要するのが「文字」,「Python 関数」および「Native 関数」というデータ型である．

- 整数
- 浮動小数点数
- None
- Python 関数
- native 関数または C 関数
- 文字 (∗)
- 文字列 (∗)
- タプル (∗)
- リスト (∗)
- 辞書 (∗)

なお，∗がついているものは，縮小仕様 (文法において，sub-Python と称したもの) では必要がないものである．

文字, Python 関数, native 関数以外: については，それらがなんのことかは説明の必要はないはずである (忘れた人はチュートリアルを参照したり，実際の Python 処理系を動かして復習するとよい)．

文字: Python には「文字」というデータ型はなく，長さ 1 の文字列で代用する．しかし処理系内部では文字 (長さ 1 の文字列) を，それ以上長い文字列と区別して表現しておくと良い．

Python 関数: は，`def` 構文を使って定義された関数のことである．Python では，関数を変数に代入したり，他の関数に引数として渡したり，他のデータ構造 (リストやタプル) に格納したりすることができる．例えば，

```
def f(x):
    return x + 1
```

のような関数定義を Python 処理系に向かって打ち込んでみよう．

```
Python 2.3.4 (#53, May 25 2004, 21:17:02) [MSC v.1200 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> def f(x):
...  return x + 1
...
```

こうすると，`f(3)` のようにして，`f` を呼び出すことができるのは当然のことだが，`f` 単独でも「関数」としての値を持っている．

実際，`f` の値を表示すると，

```
>>> f
<function f at 0x00A1D770>
```

のように表示される．関数自身の定義が文字列として出てくるわけではないが，少なくとも `f` が関数である (整数やその他のデータではない) ということはわかる．

`f` は値だから，変数に代入することができる．

```
>>> g = f
```

これで，`g` という変数に，`f` の値が代入されたことになり，`f(x)` も，`g(x)` も同じ結果を返す．

```
>>> g(3)
4
```

また，g を表示させると，f と同じものを表示することに注意しよう．

```
>>> g
<function f at 0x00A1D770>
```

名前は g ではなく，f である．つまり，g は変数で，そこには def f(x)... で定義された f の値 (関数) が代入されているのである．

当然，f をデータ構造 (例えばリスト) に入れることもできる．

```
>>> l = [ f ]
>>> l[0](3)
4
```

2 行目は f をリスト l から取り出して，そして呼び出している．

native 関数: これも通常の関数の一種だが，mini-Python のプログラムによって (つまり def によって) 定義された関数ではなく，インタプリタの内部で，C によって定義されているものである．その関数を使う (呼び出す) 際は，通常の Python 関数と同様の文法で呼び出す．

なぜこういう仕組みが必要なのか説明しておこう．例えば我々の mini-Python 処理系にファイル I/O やネットワーキング (ソケット) などの機能を追加したいとしよう．C のプログラムであれば C 言語から呼び出せるライブラリ (fopen, fread, fwrite, socket, etc.) が標準的に用意されているのでそれを呼べばよいわけだが，mini-Python にはもともとそういう機能は用意されていない．

ではそれを def を使ってい定義できるかと言うと，Python の文法の内部で何を書こうと，それは当然のことながら書けない．どこかで Python の「外へ飛び出す」方法が必要である．その仕組みが native 関数で，Python プログラムからある native 関数を呼び出すと，その実行は，処理系の内部で定義されているある関数の呼び出しに変換されるというものである．この仕組みを用意しておけば，新しい機能の追加が容易になる上，処理系に組み込みの機能 (+ や%などの演算子) も見通しよく作ることができる．詳しくは後述する．

3.2 データ表現の概要

表現しなくてはならないデータの種類がわかったところで，それらの実際の表現方法を決めよう．

最初におさえておくべきポイントは，Python ではひとつの変数にあらゆる種類のデータを格納することができるということである．たとえば以下は (C ではありえないが Python では) 合法なプログラムである．

```
x = 10
print x
x = 20.3
print x
```

```
x = [ 1, 2, 3 ]
print x
```

mini-Python の処理系内部では，この x の値はどこかになんらかの形で格納されているわけで，それが C 言語のある型であらわされていないといけないわけである．つまり，C 言語のある**ひとつの**型で，mini-Python で可能な**あらゆる**値が表現できなくてはならないのである．

これを行う際の常套手段は，C 言語の共用体 (union) を使うことである．より正確には，データの種類をあらわす tag (整数) と，tag に応じてその値を表す union をひとまとめにしたような構造体 (struct) をつくり，それへのポインタとして値をあらわす．

```
/* この py_val_t (py_val へのポインタ) が,
   処理系内部で mini-Python の値を表す型 */
typedef struct py_val * py_val_t;

enum {
  py_type_float,
  py_type_string,
  ...
} py_type_t;

typedef struct py_val {
  py_type_t type; /* データの種類 */
  union {
    /* データの種類に応じた中身 */
    double f;     /* 浮動小数点数の場合 */
    ...;
    ...;
  } u;
} py_val;
```

最も単純にはこの原則を文字通り「すべての」データ型に適用することもできるが，実はそれだと実行時のコストが大きい (つまり処理系が非常に遅くなる) のでもう一工夫したほうが良い．それがなぜかを説明する．すべてのデータ型に適用すると，最もよく使われるデータ型である整数も，以下のようにあらわされる．

```
/* この py_val_t が，処理系内部で mini-Python の値を表す型 */
typedef struct py_val * py_val_t;

typedef struct py_val {
  py_type_t type; /* データの種類 */
  union {
    /* データの種類に応じた中身 */
```

```
    int i;        /* 整数の場合 (*) */
    double f;     /* 不動小数点数の場合 */
    ...;
    ...;
  } u;
} py_val;
```

これが意味するところは，mini-Python の整数 (たとえば 5) をあらわすために，どこかに py_val 型を保持するだけのメモリが割り当てられ，その中のどこか (上記の i というフィールド) に 5 が格納され，5 自体はそのメモリブロックへのポインタとして表されるということである．

そのメモリはどのようにして確保されるかというと，一般的には整数を結果とする演算 (たとえば足し算) が行われるたびに，malloc などの一般的なメモリ割り当て操作によって，ヒープから割り当てるしかない．ヒープからのメモリ割り当ては関数呼び出しを含め，平均して数十から百命令近くはかかってしまうので，これは足し算一回を行うためのコストとしては大きい．[1]

メモリ割り当てをある程度避けるための簡単な効率化として，よく現れるであろう整数 (たとえば -100 から 100 くらいまで) をプログラム開始時にあらかじめ割り当てておき，演算を行った結果がその範囲に収まっていれば常にその領域を利用する，などの手も可能であるが，C で整数を直接扱うのには遠く及ばない．

これは，複数の型の値を保持できる変数 (多相型の変数という) を持つプログラミング言語では共通の問題である．そこで以下に可能な手法をひとつ述べることにする．

C プログラミングに不慣れな人，または一般にコンピュータ内部でデータがどのように表現されているかにあまりなじみのない人は特に注意して，以下の説明をよく理解してほしい．

基本は，

- ほとんどのデータは上記のとおり，割り当てられたメモリへのポインタとして表現される．
- 整数など，よく使われる一部のデータは，ポインタ型 (ここでは py_val_t) の変数に，無理やり整数 (など) を代入することで保持する．しかしポインタと混同しないように，以下で述べるようなタグ付けを行う．

C 言語の表面でどの Python 値も py_val_t 型の値で表されるという基本的枠組みには変更はない．違いは Python の整数を表すのに，その整数を直接 py_val_t 型の値として表現してしまうということである．早い話が (後に若干修正するのだが)，Python 整数の 5 は，処理系内部の py_val_t 型の変数やフィールドに直接 5 が代入されることで表現されるわけである．ポインタ型の変数に直接 5 を代入するというのは乱暴な話であるが，これが可能な理由は，資料 (4) の 8 節: コンテナでも述べたとおり，ポインタ型の正体は単なるメモリアドレス (番地) であり，機械語レベルでは整数と同じものだからである．

ただしこの仕組み—整数 i を表現するのに py_val_t に i を代入する— と困るのは，データが整数なのかポインタなのかを区別する手段がなくなることである．つまりたとえば，文字列を表現するためにあるメモリブロックを割り当て，そのアドレスが 10000 番地であった場合，py_val_t 型の

[1] もっとも我々の処理系で足し算一回を行うのにかかるコストはその他にも様々なものが存在するので，これがどの程度処理系全体の性能に影響を与えるかは探ってみる価値がある．

変数やフィールドに格納された 10000 が，メモリ番地のことであるのか，整数の 10000 であるのかを区別する手段がなくなる．

そこで我々は,「通常のメモリ割り当て器 (malloc など) が返す値は，4 の倍数とほぼ決まっている」という事実を使う．こうなっている理由は後で述べるとして，それによればメモリ割り当て器が返してくる値は，常に下位 2 bit が 00 になっている．そこで整数を表現するのに，下位 1 bit を 1 にし，残りの 31 bit 部分に表したい整数を格納することにする．

メモリ割り当て器 (malloc) が返すアドレス　メモリ割り当て器 (malloc や new) を使って返された値 (つまりメモリ上のアドレス) がどんな整数であるのかに関心を持った人は少ないかもしれないが，今の我々には重要である．特に，現代の計算機でメモリ割り当て器が返すアドレスはほとんどの場合 4 ないし 8 の倍数になっている，という事実が重要である．我々の処理系では 4 の倍数であることを仮定し，8 の倍数であるという仮定まではしない．

もちろんなぜそんなことになっているかには理由がある．ほとんどの CPU は整数 (int) や浮動小数点数 (double) をアクセスするために，4 byte や 8 byte 単位のメモリ読み書き命令を備えている．そして，それらはそのアドレスが 4 や 8 の倍数になっているときに速く (少ないサイクル数で) 実行されるようになっていることが多い．CPU によってはそうなっていないと例外が発生するものもある (SPARC)．そこでほとんどの場合コンパイラは整数や浮動小数点数を，適切な倍数の番地へ配置しようとする．そのための前提として，メモリ割り当てライブラリは 4 や 8 の倍数のアドレスを返すように作られるのが普通なのである．

仮にメモリ割り当て器が 4 の倍数を返してこなかったらどうすればよいのか? というのは良い疑問である．心配ならば，返してきた値を下回らない最も近い 4 の倍数を使うようにすれば良いのである．ただしこれをしたければ，メモリ割り当て器にメモリを要求する際に，3 バイトほど余計にメモリを要求する必要がある．事実上，ここまで心配する必要はない．

3.3　データ表現の詳細

以上の前口上の元，ようやく実際のデータを表現する約束を書いていこう．

3.3.1　整数

mini-Python では，整数を 32bit の bit 列の上位 31bit のみを使って表す．当然のことながら表せる整数の範囲は,

$$[-2^{30}, 2^{30} - 1]$$

となる．では最下位 bit には何を入れるのかというと，常に 1 を入れる．この，最下位 bit に 1 がくることによって，整数の印とするのである．このようなデータ型を区別するために使われる bit (あるいは数 bit) のことをしばしば,「タグ」という．

今，整数は最下位 bit を 1 にすると決めたので，他のデータの最下位 bit は 0 にしなくてはならないことに注意．

さて，以上は bit 列という，いわば機械語の言葉を使って述べられたデータ表現だが，皆はこれを無事，C 言語のプログラムで，それも

あらゆるPython値は，評価器内部では，py_val という構造体へのポインタ型 (py_val_t 型) として表現される

という約束に従って表現できるだろうか? つまりたとえば，mini-Python の “5” を表す C の「式」を上の約束にしたがって書けるか，と言うことである．その答えは，

```
((py_val_t)((5 << 1) + 1))
```

である．(5 << 1) によって，通常の 5 の 2 進法表現を 1bit 左にシフトし，1 を足すことでタグをつけている．つまり C のプログラム内では 11 ということである．それも無理矢理ポインタ型の値にするためにキャストをしているから,「11 番地」というアドレス (ポインタ型の値) によって 5 を表しているのである．

これを以下のような関数として定義しておこう．

```
py_val_t mk_py_int(int i)
{
  return (py_val_t)((i << 1) + 1);
}
```

ためしに，py_val_t 型の値として与えられた二つの整数の足し算をして，結果をやはり py_val_t 型の値として返すような関数を書いてみよう．

```
py_val_t py_int_add(py_val_t a, py_val_t b)
{
  int a_ = (int)a;
  int b_ = (int)b;
  return mk_py_int((a_ >> 1) + (b >> 1));
}
```

でよい．ただしこれは，与えられた a や b が，実際 Python の整数であるか (つまり，a や b の最下位 bit が 1 か) を検査していない．実際のインタプリタではその検査は必須であり，上はあくまで説明用のコードである．

3.3.2 文字

文字は C 言語では 8bit の bit 列で表されるデータである．したがって整数とも，(4 の倍数である) メモリアドレスとも (そしてこれから決めるほかのデータとも) 区別できるようなタグをもちいれば，1 文字を，タグを含めて 32bit で表すのは簡単である．最下位 1 bit が 1 (つまり ···1) だと整数．最下位 2 bit が 00 (···00) だとメモリアドレスであったから残されたスペースは最下位 2 bit が 10 (···10) というものである．文字以外にも若干のスペースを残しておく必要があるから，我々は文字を表すタグとして，最下位 3 bit が 010 (···010) とすることにする．つまり，’a’ という文字であれば,

```
(py_val_t)((((int)’a’) << 3) + 2)
```

という値として表現される (2 はいうまでもなく 010 を整数に直したもの)．

3.3.3 None と空文字列

ここまでで，最下位 3 bit が 110 というスペースがまだ使われていない．

したがって None は 0..000110 (つまり 6), 空文字列は 0..001110 で表現する．最下位 3 bit が 110 でその上位 3 bit が 000, 001 以外，というスペースがまだあいていることになる．

3.3.4 浮動小数点数

我々の決断である,「すべての値を 32bit の bit 列 (py_val_t) として表す」ことで，一番困るのは，浮動小数点数をどう表すかである．浮動小数点は計算機の内部では 64bit, 80bit などの，整数よりも長い bit 数で表される．[2]実際機械語レベルでも，浮動小数点演算をするためのレジスタと，整数演算やメモリアクセスのアドレスとして使われるレジスタはほとんどすべての CPU において，別々のレジスタである．Pentium はレジスタ上では 80bit, ただしメモリに格納するときは 64bit という (珍しい) 方式を用いている．Pentium 上の C 言語では double という型が 64bit の浮動小数点数を表す型であるが，実際に演算しているときは (こっそり)80bit の演算をしていることもあるわけである[3].

C 言語において，整数を浮動小数点数の変数に代入したり，その逆を行うことも可能だが，これはポインタと整数間の代入のように，bit 列がそのまま移動しているだけではない．そもそも bit 数が違うのだからそのようなことはしようがない．その代わりに，整数 x を浮動小数点数の変数に代入する場合，x と同じ数を浮動小数点数にしたもの (例えば x が 5 なら 5.0) に変換されて代入される．これは bit 列としてはまったく違うものである．逆に浮動小数点数を整数の変数に代入するときは，整数への丸め (例えば $5.3 \to 5$) が行われる．つまり値として等しくすらない値へ変換されるのである．

いずれにせよ，64bit ないし 80bit を要する浮動小数点数を 32bit で表すのは無理な相談である．やむをえないので，最も最初のアイデアであった py_val 構造体の中の共用体の 1 フィールドとして double 型のフィールドを用意し，そこへ格納するのである．

メモリは必要になるたびに malloc を用いて確保する．言い換えれば，浮動小数点数同士の足し算を実行するのに，新しい領域を malloc する．整数のときと同様，C の double の値から Python 値の浮動小数点数を作る関数と，ためしに二つの Python の浮動小数点数の足し算をする関数を書いてみよう．

```
/* 浮動小数点数 f に対応する Python 値を作り出す */
py_val_t mk_py_float(double f)
{
  py_val_t r = alloc_boxed(py_type_float);
  r->u.f = f;
  return r;
}
```

[2]正確には 32bit の浮動小数点数も存在するがそれを使えばここで述べる問題がなくなるわけではない．

[3]80bit で演算をしているかどうかは C のプログラムから予測できないので，ある意味ではたちが悪い

ここで alloc_boxed はメモリを割り当て，型を表すフィールドをセットする，以下のような関数である．浮動小数点以外でも多用される．

```
py_val_t alloc_boxed(py_type_t type)
{
  py_val_t r = (py_val_t)my_malloc(sizeof(py_val));
  r->type = type;
  return r;
}
```

構造体 py_val 中には浮動小数点数に限らず，32bit で直接表現できない様々なデータがおかれる．例えば辞書，リスト，タプル，文字列などは明らかにこの範疇に属するし，実は Python 関数や native 関数も，こうして必要な情報 (Python 関数の名前や定義) を格納する．それらのデータ型はすべて py_val が置かれているアドレスとして表される．それらの型の間では，型の区別は py_val のフィールドである type によって行われることに注意.

3.3.5 文字列

文字列は C の char*型へのポインタに加えて，長さをしめす整数を構造体に格納して表すことにしよう．それにともなって py_val を拡張すると以下のようになる．

```
typedef struct py_val * py_val_t;

typedef struct py_string
{
  int n;          /* 長さ（文字数） */
  char * a;       /* 文字列へのポインタ */
} py_string, * py_string_t;

typedef struct py_val
{
  py_type_t type;
  union {
    double f; /* 浮動小数点数 */
    py_string s; /* 文字列 */
    ...                          /* other cases ... */
  } u;
} py_val;
```

3.3.6 Python 関数

Python 関数は，(def によって定義されたときの) 名前，引数の名前 (文字列のベクタ)，そして本体である文のベクタからなる構造体へのポインタで表現できるだろう．そしてもちろん，他のデータと区別するために，py_val の中に格納する．つまり，

```
typedef struct py_ifun
{
  char * name;                    /* 関数名 (def されたときの名前) */
  str_vec_t ps; /* 引数のベクタ */
  stmt_vec_t b; /* 本体 (文のベクタ) */
} py_ifun, * py_ifun_t;

typedef struct py_val
{
  py_type_t type;
  union {
    double f; /* 浮動小数点数 */
    py_string s; /* 文字列 */
    py_ifun i; /* Python 関数 */
    ...                           /* other cases ... */
  } u;
} py_val;
```

3.3.7 Native 関数

Native 関数は C 関数へのポインタが主なデータだが，Python プログラムから呼ばれたときに引数の数などを検査するために付随する情報を構造体に格納する．

```
/* C_fun_t は「py_val_t を返す関数の型」を定義する */
typedef py_val_t (*C_fun_t)();

/* Native 関数 */
typedef struct py_nfun
{
  char * name;         /* 名前 (print やエラーメッセージのため) */
  int arity;           /* 引数の数 */
  C_fun_t f;           /* C 関数へのポインタ */
} py_nfun, * py_nfun_t;
```

3.3.8 タプル，リスト

タプルとリストはともに，他の Python 値を格納するためのいわゆる「コンテナ (入れ物)」データ構造であって，以下のような共通する操作を持っている．以下で a はタプルまたはリストである．

- 生成する (タプルは (a, b, c), リストは [a, b, c] のように)
- 整数 i を与えて i 番目の要素をアクセスする (`a[i]`)
- 要素数を数える (`len(a)`)
- (リストのみ) append (`a.append(x)`)
- (リストのみ) 整数 i を与えて i 番目の要素へ代入 (`a[i] = x`)
- (リストのみ) 整数 i を与えて i 番目の要素を削除 (`del a[i]`)

両者は実際良く似ていて，タプルはリストの部分集合のようにすら見える (つまりタプルは必要ない)．だから処理系内部ではほとんど共通のデータ構造で両者を実装することもできる．

実際には，両者に期待される性能の性質は若干異なり，したがって実装もそれに応じて異なるものを使うほうが，性能のためには望ましい．その性質とは，

- リストは append を O(1) のコストで実行することを期待される
- 一方リストは，i 番目の要素へのアクセス (読み出し，代入，削除とも) は $O(i)$ で実行できればよい (つまり遅くてよい)
- タプルは代入や append ができない代わりに，要素へのアクセスは，配列同様 $O(1)$ のコストで行えることが期待される．

したがってリストはいわゆるリンクリスト (1 要素だけを格納したデータ構造をポインタでつないだもの) で表現され，タプルは生成時に要素数が決まるので，要素を配列にしまうデータ構造が用いられる．もちろんその両者をうまく折衷して，(短い) リストのアクセス時間を短縮する，などの工夫が考えられる．

ここは各自，データ構造やアクセスのアルゴリズムの試行錯誤をしてもらいたい (といっても，まずは単純なものをつくり，処理系がいったん完成してからチューニングをするのが良いだろう)．

3.3.9 辞書

辞書もリストと似ているが，添え字として整数に限らないものが許されているところが非常に特徴的である．以下では a を辞書オブジェクトとしている．

- 生成する (`{}`, `{ 1 : 2 }`のように)
- キー k を与えて対応する要素をアクセスする (`a[k]`)
- 要素数を数える (`len(a)`)

- キー k を与えて値を代入 (`a[k] = x`)
- キー k に対応する値を削除 (`del a[k]`)
- 全キーをリストにして返す (`a.keys()`)
- 全値をリストにして返す (`a.values()`)
- 全てのキー：値のペアを，タプルのリストにして返す (`a.items()`)

辞書は概念的には，キー：値という組み合わせを多数内部に保持しているデータ構造である.「多数内部に保持」する方法はもっとも単純にはそれらを適当に配列に格納しておけばよい．しかしこれは，キーを与えられて対応する要素を取り出すのに，毎回その配列を線形探索しなくてはならず非常に効率が悪い.

辞書オブジェクトに期待されている性能は，ハッシュ表を使うことによって，要素へのアクセスがほぼ $O(1)$ でおわるということである.

まずは単純なものを作って話を前に進め，後から性能を測定しながらチューニングして行くことを進める.

4 今週の課題

データ表現の方法を理解し，`pyvalues.h` と `almost_empty_pyvalues.c` をダウンロードする.
そこではだいたい以下のものが含まれている.

- `py_val_t` の C のデータ型としての定義
- 各種データ型について，そのデータを (正しい `py_val_t` 型の値として) 作る関数 (`mk_py_int`, `mk_py_float` など) の定義
- 一部のデータ型について，`py_val_t` 型の値がそのデータ型であるかどうかを判定する関数 (`py_is_int`, `py_is_float` など) の定義

例によって枠組みだけを提供し，肝心なところには穴を開けてある．今回はおもに，データ表現を実際に扱う部分に穴を開けている．それらについて，必要であれば補助的なデータ構造を実装し，各種操作を実装せよ.

まずは単純なものを早く実装し，後に色々なプログラムが動くようになってから，性能向上を目指して変更していく，という心構えでいること.