B演習 (言語処理系演習) 資料 (1)
仕様説明，環境設定，Python の練習

田浦

2006/10/16

1 課題概要

本演習では，mini-Python と呼ばれる言語の処理系 (インタプリタ) を作る．世の中で使われているポピュラーな言語に Python というものがある．mini-Python はその仕様を演習用に縮小したものである．

以下の演習 HP で，随時課題のアナウンス，補足説明，各種情報へのポインタなどを提供する．

`http://www.logos.ic.i.u-tokyo.ac.jp/lectures/enshu2006/`

2 Python について

2.1 情報源

オリジナル: http://www.python.org/

日本語: http://www.python.jp/

各種フォーマットでのドキュメント: https://sourceforge.jp/projects/pythonjp/files

推奨フォーマット:

印刷用: pdf

オンライン閲覧 (Windows): chm

オンライン閲覧 (Linux): html

まずはチュートリアルをダウンロードして，5 章までに目を通すこと．

mini-Python は大雑把に言って，このチュートリアルの 3-5 章に書かれている部分に対応している．

2.2 Python 処理系

後々の性能比較や動作検証のために，Python の処理系を使えるようにしておくこと．

Linux 環境であればほぼ確実に，すでにインストールされているはずである．されていない場合のインストール方法は，付録 A 節を参照.

無事インストールされていれば，python と打ち込むことで処理系が立ち上がって以下のようなメッセージを表示する．

```
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.
>>>
```

チュートリアルなどを見ながら，簡単な式を打ち込んでみよ．

3 今週の課題

- 班を作る
- 教室からインターネットへ接続
- 近山・田浦研サーバ (python.logos.ic.i.u-tokyo.ac.jp) へログイン．パスワード変更．ssh のキー設定 (演習 HP 参照)
- Python に慣れるための練習問題

4 Python 練習問題

以下を，チュートリアルとライブラリリファレンスを読みながら書いてみよ．最後の二つを除けばすべて 10 行程度で書けるはずである．途中で Python に慣れてきて馬鹿馬鹿しくなったら，途中を飛ばして，最後のチャレンジ問題をやるとよい．

モジュールの使い方: string モジュールを import して，以下の関数の動作をマニュアルと処理系で確かめてみよ．`find`, `split`, `join`, `strip`.

標準入出力: sys モジュールを import して，以下のような関数 hello を作れ．いくつか細かい落とし穴がある．

- `"hello, your name? "`と表示
- 標準入力から名前を読み込む
- `"hi `〈入力された名前〉`, nice to see you"`と表示.

例:

```
>>> hello()
hello, yourname? Hisashi
hi Hisashi, nice to meet you
```

基本的な file I/O: 組み込み関数 `open`, ファイルオブジェクトのメソッド `read`, `readline`, `readlines` の動作をマニュアルで調べよ．そして，以下のような関数を書いてみよ．

- `cp`(*src*, *dest*)：*src* の内容を *dest* へコピーする
- `wc`(*filename*)：与えられたファイルのバイト数，単語数，行数を表示する．「単語」とは，単にスペースまたはタブを含まない最大の文字列のことであるとする (ヒント: string.split)．

辞書: 与えれたファイル中の単語のヒストグラム (どの単語が何度出現したか) を表示するプログラムを書け．辞書オブジェクトを使うと，単語の出現回数を簡単に保持できる．

簡単な応用: 与えられた数 n 未満の素数を小さい順に並べたリストを返す関数．n をいくつくらいまで伸ばせるか? 例:

```
>>> primes(10)
[2, 3, 5, 7]
>>> primes(100)
[2, 3, 5, 7, ..., 97]
```

チャレンジ: 3x3 のいわゆる○×ゲームの解 (両者がベストを尽くしたときに，先手必勝，後手必勝，必ず引き分け，のどれになるか) を探索を用いて求める．

5 注意を要する言語仕様

同等比較演算子: Python である二つのデータが「同じ」かどうかを比べるのに，`==`, `is` の二つがある．非常に直感的に言えば，前者は両者が「値として同じ (表示したときに同じに見える)」かどうか，後者は「両者が同一の “もの” であるか」を調べている．

整数や文字列など，更新できないデータの場合，両者の区別は存在しない．両者の区別は以下のような例で明らかになる．

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a == b
True
>>> a is b
False
```

また，

```
>>> c = a
>>> a is c
True
```

である．a is bが成り立てば，a == b も成り立つ．

変数のスコープ: Python の変数には，局所変数と大域変数の二つがある．C/C++を知る人には聞きなれた概念だと思うが，文法の違いから，多少違いが生ずるので，注意が必要．

局所変数: ある関数の中で代入された変数は,「局所変数」(ローカル変数とも言う) であり，その関数内でしか参照できない．

```
>>> def f():
...     x = 10
...     print x
...
>>> f()
10
>>> print x          # x はあくまで f() の中からしか参照できない
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'x' is not defined
>>> def g():
...     print x
...
>>> g()              # x はあくまで f() の中からしか参照できない
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in g
NameError: global name 'x' is not defined
```

また，実行中のある時点で同じ関数が複数回呼び出されていても (たとえば再帰呼び出しで)，それぞれの呼び出し中で使われる局所変数は (記憶場所としては) 異なるもので，それぞれの関数呼び出しに固有の値を保持する．例:

```
>>> def f(n):
...     x = n
...     if n > 0: f(n - 1)
...     print "x = %d" % x
...
>>> f(3)
x = 0
x = 1
```

```
x = 2
x = 3
```

つまり，f(3) → f(2) → f(1) → f(0) という再帰呼び出しの連鎖の中で，x の値はそれぞれの呼び出しの中で別に記憶されている．

大域変数: C 同様，関数定義の外側で代入された変数は,「大域変数」(グローバル変数とも言う) と呼ばれ，どの関数中からでも参照できる．

```
>>> y = 10
>>> def f():
...     print y
...
>>> f()
10
>>> def g():
...     print y + 1
...
>>> g()
11
```

局所変数は大域変数に優先する．詳しく言えば，y を局所変数として定義した (つまり y へ代入をした) 関数内では，y は局所変数の方を優先的に参照する．例:

```
>>> y      # 大域変数 y の値は 10
10
>>> def h():
...     y = 20
...     print y
...
>>> h()
20
>>> y      # 大域変数 y の値は 10 のまま
10
```

まとめると，変数の参照は，まずそれがその関数内で局所変数として定義されていればそれを参照し，なければ大域変数を参照し，それもなければエラーになる．[1]

関数の中での大域変数への代入: ここで鋭い人は，大域変数の値を，関数の中で変更するにはどうしたらいいか，疑問に思うことだろう．上の h の例で，y = 20 は大域変数の y を変更するのではなく，h 中に局所変数 y をつくり，その局所変数を変更したのだった．代入文を書くと常にこのように解釈されるのであれば，関数の中で大域変数の値を変更する手段はないことになる．Python ではこれに対する逃げ道として，

[1] 実は最近のバージョンの Python の規則はもう少し複雑だが，ここでは述べた通りの規則を採用する．

global 〈変数名〉

という構文 (宣言) を用意している．これは関数中に書くことで,「以降この関数内では, その変数名は常に大域変数の方を意味する」という効果を持つ．したがって，大域変数を更新するには以下のようにすればよい．

```
>>> def update_global():
...     global y
...     y = 100
...
>>> y
10
>>> update_global()
>>> y
100
```

6 mini-Python

6.1 実装すべき範囲

課題で作る mini-Python の処理系がカバーすべき範囲の説明に入る．以下の説明は，Python という言語の輪郭はつかんでいることが前提である．なお，いずれ構文解析器を作る際に，mini-Python の文法を厳密に定義する．実装すべき範囲の明確な線引きはそこで完全に明らかになる．

6.1.1 文

- pass
- return (e.g., return 15)
- continue
- break
- del (e.g., del a[x])
- print (e.g., print “hello”)
- global (e.g., global x)
- if
- while
- for

- def
- 式文 (statement expression)

6.1.2 データの種類

- 整数
- 浮動小数点数
- 文字列
- None
- タプル
- リスト
- 辞書

特に，本物の Python においては重要な，オブジェクトは必須としない．これを省略することで mini-Python の言語仕様はずいぶんと小さくなっている．

6.1.3 式

- 変数 (e.g., x)
- リテラル (e.g., 3, 3.5, “abc”).
  - 整数
  - 浮動小数点数
  - 文字列
  - None
- 複合リテラル (display) (e.g., [1, 2, 3], (1, 2, 3), 1 : “a”, 2 : “b” , [(1, 2), f(x), [3, 4]])
- 演算子 `or` および `and`
- `or` および `and` 以外の各種演算 (e.g., `1 + 2`, `x is y`, `f(x) == g(x)`). 演算の種類は以下のとおり．

  `or`, `and`, `not`, `<`, `>`, `==`, `>=`, `<=`, `<>`, `!=`, `is`, `is not`, `in`, `not in`, `|`, `^`, `&`, `<<`, `>>`, `+` (2 項および単項), `-` (2 項および単項), `*`, `/`, `%`, `~`,

  `or` と `and` を特別扱いする理由は，評価方法の違いによる．
- リスト，タプル，辞書からの要素選択 (e.g., `a[x]`)

- 関数呼び出し (e.g., `f(x, y, z)`)．ここで f はユーザがプログラムの中で定義した関数の場合と，処理系にもともと組み込まれている関数の場合がある．
- メソッド呼び出し (e.g., `a.f(x, y, ...)`) ただし，mini-Python ではこれは純粋に `f(a, x, y, ...)` の略記とする．

mini-Python が，大雑把に Python から何を削ったものであるのかは，付録 B 節に述べておいた．また，実際に処理系を作る際には，さらに上の文法を簡単化するための手助けを提供する．

6.2 最終提出作品

実行形態は，

> コマンドラインでファイル名を受け取って，そのファイルを mini-Python プログラムとして実行する

という形態を必須条件とする．一斉テストなどをやりやすくするために，コマンド (実行可能ファイル名) は，`minipy` (Windows であれば `minipy.exe`) で統一する．まとめると，以下のようなコマンドラインを受け付けることを必須とする (prompt クトリに作成した実行可能ファイルがあるものと仮定する)．

```
prompt% ./minipy ファイル名
```

これを行うと，ファイル名の内容を順次実行する．

もちろんこのほかにいろいろなコマンドラインオプションを許すのは自由である．

対話的に式や文を受け付けてその結果を表示する，という形態は必須とはしないが，ファイル名が特別な名前 (よく使われるのは，`-`) であったときに，プログラムをファイルの変わりに標準入力から読み込む，という動作は便利で，デバッグもしやすくなる．

A　Unix 環境での標準的なソフトウェアのインストール方法

A.1　Linux のパッケージ管理ツール apt

Linux や BSD などの OS では，多くのソフトウェアが，パッケージ管理ツールを用いてたちどころにインストール可能である．Linux でも OS によってコマンドが異なるが，Debian や，演習用に提供されている Vine Linux では apt と呼ばれる非常に使いやすいツールが提供されている．必要最低限の覚えるべき操作は，

```
% su          # スーパユーザになる
# apt-get install〈パッケージ名〉
```

パッケージ名はインストールすべきソフトウェア名で，Python であれば，python．多くの場合容易に想像できる名前である．

パッケージ名がわからないときに検索をするには，

```
% apt-cache search〈検索キーワード〉
```

で，パッケージの説明中に検索キーワードがあらわれるパッケージを検索できる．

A.2　ソースからのコンパイル

Windows では多くのソフトウェアは自己解凍形式のインストーラによって提供されている．大概の場合，そのインストーラを実行して，どこにインストールするかなどの質問に答えるとインストールが完了する．

Linux や FreeBSD の場合，apt-get や rpm などのパッケージ管理ソフトによって簡単にインストールできるソフトウェアも存在するが，ソースからコンパイル，インストールする必要がある場合も多い (パッケージとして提供されていなかったり，インストール場所を変更したい，など)．

ソースからコンパイルする場合，そのやり方はソフトウェアごとに付属するドキュメント (README や INSTALL という名前のファイルが付属することが多い) を参照するのが基本だが，実際の手順は多くの場合共通である．Unix での模範的なソフトウェアは，多くの場合以下の手順でインストールできる．以下では，ダウンロードしてきたソースパッケージが，app.tar.gz であり，これを解凍すると，app というディレクトリができると仮定してある．

```
prompt% tar zxf app.tar.gz      # tar.gz を展開（app ディレクトリができる）
prompt% cd app                  # できたディレクトリに移動
prompt% ./configure --prefix <インストール先ディレクトリ名>
prompt% make
prompt% make install
```

4 行目の make はプログラムをコンパイラによってコンパイルし，プログラムファイルやライブラリなどが作られる．この時点ではそれらのファイルは，展開したディレクトリ内にのみ存在し，インストールが行われないのが普通である．5 行目 make install が，実際にできたプログラムファイ

ルやライブラリをインストール (所定の場所に移す) コマンドである．それがどこであるかは，3 行目の<インストール先ディレクトリ名>で指定される．たとえば，<インストール先ディレクトリ名>として/home/tau/local を指定すると，プログラムファイルは，/home/tau/local/bin へ, ライブラリの類は/home/tau/local/lib へ，インストールされるのが慣習である．./configure を実行する際，--prefix を指定しないと，/usr/local を指定したとみなされるのもまた，多くの場合に用いられている慣習である．

通常のユーザ権限では/usr/local に書き込みが禁止されている場合が多いので，/usr/local へインストールする場合には確認が必要である (ls -l /usr/local など)．make install の代わりに，make -n install を実行すると，インストール時に実行されるコマンドを表示し，実際のインストールは行わない．こうすることで，コマンドがどこへ書き込まれるかを事前にうかがい知ることができる．

以上は，python の処理系を含め，多くの「標準的・模範的」ソフトウェアが用いている慣習であるが，あくまで慣習に過ぎないので，実際のインストール手順については各ソフトに付属するドキュメントを参照すべきなのはもちろんである．

A.3 環境変数 PATH

インストールされているはずのコマンドをプロンプトに打ち込んで,「command が見つかりません」という旨のメッセージが表示されたら，それは，そのコマンドが存在するディレクトリが PATH 環境変数に含まれていないことが原因である．

```
% echo $PATH
```

で，PATH 環境変数を表示できる．ここに，目当てのコマンドが存在するディレクトリ (たとえば目当てのコマンドが/home/tau/local/bin/python であれば，\verb/home/tau/local/bin+) が存在するかどうかを確かめればよい．

環境変数を変更するには，Windows XP であれば，マイコンピュータを右クリック → プロパティ → 詳細設定 → 環境変数で，PATH という名前の変数 (必ずあるはず) を見つけ，そこに目当てのコマンドが存在するディレクトリを追加する．

Unix では，自分の使っているシェルに対応する設定ファイル (bash なら．bashrc, csh なら.cshrc, など) 中で PATH 環境変数を設定する．設定の仕方は，bash なら，

```
export PATH=/usr/local/bin:$PATH
```

など (python が/usr/local/bin にある場合)．csh では，

```
setenv PATH /usr/local/bin:$PATH
```

自分がどのシェルを使っているか分からない場合，

```
echo $SHELL
```

と打ち込めばわかる．追加したら csh/bash を改めて起動するか，

```
. ~/.bashrc
```

```
source ~/.cshrc
```

などとして設定ファイルを読み込みなおそう.

B mini-Python と Python の主な違い

B.1 class 定義とオブジェクト

Python はオブジェクト指向言語である．C++, Java などのオブジェクト指向言語と同様，Python には class 定義構文があり，これを用いて新しいデータ型を定義することができる．また，class には関数 (オブジェクト指向の用語では通常メソッドと呼ぶ) の定義を付随させることができる．この際に，同じ名前の関数でも class が異なれば，別の関数とみなされる．class を定義すればその class に属するデータ (通常オブジェクトと言う) をいくらでも作ることができる．オブジェクトにメソッドを呼ぶことができ，そのオブジェクトがどの class に属しているかによって，適切な定義が参照されて実行される．

mini-Python は，class 定義を許可しないことにする．結果として「新しいデータ型」を作る機能がなくなるので，プログラム実行中に現れる (作られるデータ型) は言語仕様で定められた，固定された種類 (具体的には，整数，浮動小数点数，文字列，タプル，リスト，辞書，None，関数) のみである．これは実装を大幅に単純化する．

帰結として，オブジェクトの要素 (フィールド) を選択する構文 (e.g., `o.f`) も不要になった．ただし，メソッド呼び出し構文については少し特別扱いが必要な場合がある．次節を参照.

B.2 メソッド呼び出し構文

Python では，メソッド呼び出し構文は，`o.f(x, y, z)` のような形をしている．これは，オブジェクト `o` が属しているクラスで定義された，`f` という名前のメソッドを呼び出すという意味である．mini-Python には class 定義・オブジェクトがないのだからこの構文自体が不要のようなものだが，少し例外がある．それは，組み込みのデータ型，リストや辞書に対しても，この構文が使われるのである．たとえば,

```
a = [1,2]
a.append(3)
print a
```

で 2 行目の `a.append(3)` はリスト `a` の最後に 3 を追加するというメソッドの呼び出しである．言い換えると，Python では組み込みのデータ型の一部はすでにオブジェクトなのである.

mini-Python では，この形のメソッド呼び出しは許さないこととし，変わりに，通常 Python でメソッド呼び出し構文 `o.f(x, y, z)` で書かれる操作は，形式的に，`f(o, x, y, z)` と書かれているものとする．つまり，`f` という名前の関数が `def` 文で定義されてあればそれを呼ぶ，ということになる.

B.3 例外

Python には Java と同様，例外処理という機構 (try, except, finally) が備わっている．例外は，プログラム中で明示的に発生させることもできる (raise 文) が，そのほかにも，0 で割り算をしたり，リストの添え字を越えてアクセスする，未定義の変数や関数を参照する，などの言語に組み込まれた操作が実行不可能な際にも発生する．例外処理はこのような事態が発生したときにそれを受け止めて，他の処理を続行させたり，または例外が発生してもクリーンに一連の処理を終了させたりする場合に用いる．mini-Python ではこれを省略する．raise 文がないことは当然の帰結であるが，上述した組み込み操作の失敗 (0 での割り算など) で発生した例外に対しては，インタプリタごとプログラムを終了させるという単純な処理を行うことにする．これによってインタプリタのコードは各所でずいぶんと簡単になる．

ただし，例外でプログラムを終了する場合は，エラーが発生した場所などを表示することが重要である．さもないと，mini-Python プログラムの開発効率は著しく悪くなる．急がば回れの例として，インタプリタに，例外発生場所を表示する機能をきちんと実装して，その後の mini-Python プログラムの開発効率を高めてもらいたい．

B.4 その他構文的なこと

その他，主としてプログラムの読み込み，文法チェック (構文解析という) を容易にするためにいくつかの省略を行うが，それはやがて mini-Python の文法を厳密に定義する際に，おのずと明らかになる．

最終的には，規定のテストプログラムを 1 文字たりとも変更することなく，正しく実行するかエラーを指摘することができるだけの構文を実装することを仕様とする．