

やり直し Python

増田 泰

日本 Python ユーザ会, 大阪大学

Agenda

- 再入門
 - データ型, プログラム, 高度なデータ表現
- 標準ライブラリの紹介
 - 正規表現, Web関連, GUI, etc.
- その他のトピック
 - 日本語の扱い, パッケージング etc.

再入門

再入門

- データ型
 - 数値, シーケンス (文字列, リスト, タプル), 辞書
- プログラムの構成
 - 制御文, 関数, モジュール
- 高度なデータ表現
 - `iterable`, 内包表記, ジェネレータ

数值型 (1/2)

1234 # 整数 (plain integer)

9876543210L # 長整数 (long integer)

True/False # Bool 型 (boolean)

1234.5 # 浮動小数点 (float)

1234.5+6789.0j # 複素数 (complex)

数値型 (2/2)

基本の演算

```
>>> 123+456
579
```

加算

```
>>> 789-987
-198
```

減算

```
>>> 654*321
209934
```

乗算

```
>>> 123/4
30
```

除算 (端数切り捨て)

```
>>> 123/4.0
30.75
```

どちらかが浮動小数点だと...

```
>>> 123.0/4
30.75
```

小数の除算になる

```
>>> 123%4
3
```

剰余

```
>>> 456**7
4099717187644686336L
```

べき乗

大きな値は自動で長整数になる

シーケンス型

"ymasuda" # 文字列型 ([unicode] string)

[1,2,"five"] # リスト型 (list)

("spam", "egg") # タプル型 (tuple)

文字列型 (1/3)

- unicode と string
 - string ... バイト列： **byte** 単位
 - Python は個々の string データの文字コードを管理しない
 - 入出力はすべて string
 - unicode ... 多言語用の内部表現： **文字**単位
 - Python 内部の文字列処理は
 - unicode 対応
ascii string 対応
 - non-ascii string 非対応

文字列型 (2/3)

4 種類の書き方

"Python\t初心者" # 通常の文字列

r"Python\t初心者" # raw 文字列

u"Python\t初心者" # unicode 文字列

ur"Python\t初心者" # raw unicode 文字列

文字列型 (3/3)

```
>>> print "spam\tegg"          # 通常の文字列
spam      egg                  # \t はタブになる
>>> print r"spam\tegg"         # raw 文字列
spam\tegg                      # \t は '\ ' と 't' として
                                # 扱われる

>>> print u"spam\tegg"         # unicode も同じ
spam      egg
>>> print ur"spam\tegg"
spam\tegg
```

文字列の操作

```
>>> 'spam' + 'egg'           # 加算は連結になる
'spamegg'
>>> 'spam' * 4                # 乗算は繰り返しになる
'spamspamspamspam'
>>> 'spam %s' %('egg')       # C ライクな文字列置換
'spam egg'
                                # 辞書も使える
>>> 'spam %(topping)s' %({'topping': 'bacon'})
'spam bacon'
```

タプルとリスト

```
>>> (1)          # これはタプルではない
1
>>> (1,)         # 末尾にカンマを打つ習慣を！
(1,)
>>> ()          # 空のタプル. カンマなしに注意！
()
>>> (1,2,"five",) # どんな値でも組み合わせできる
(1, 2, 'five')
>>> (1,("bacon","egg"),3,) # 入れ子にできる
(1, ('bacon', 'egg'), 3)

>>> [1, 2, "five"] # リストも同じ
[1, 2, 'five']
>>> [1, ("bacon", "egg"), 3]
[1, ('bacon', 'egg'), 3]
```

シーケンスの操作

```
>>> (1,3,) + (5,7,)      # 加算は連結
(1, 3, 5, 7)
>>> [2,4] + [5,6]
[2, 4, 5, 6]
>>> (1,) * 10           # 乗算は繰り返し
(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)

>>> a = [5,1,3,7]
>>> len(a)              # len() で長さを返す
4
>>> a[2]                # (0 から数えて) 2 番目の値
3
>>> a[-1]               # 末尾から 1 番目の値
7
```

スライス

```
>>> a = [1,2,3,4,5,6,7,8]
>>> a[:3]          # 3 つ目までの部分配列
[1, 2, 3]
>>> a[3:]         # 3 つ目以降の部分配列
[4, 5, 6, 7, 8]
>>> a[2:5]        # 2 つ目から 5 つ目まで
[3, 4, 5]
>>> a[2:5:2]      # 2 つ目から 5 つ目まで, 1 つおき
[3, 5]
>>> a[::2]        # 偶数個目
[1, 3, 5, 7]
>>> a[2:3] = []   # 選択範囲を空列で置き換え (削除)
>>> a
[1, 2, 4, 5, 6, 7, 8]
>>> a[:0] = [0]   # 選択範囲に挿入
>>> a
[0, 1, 2, 4, 5, 6, 7, 8]
```

*タプルや文字列は
スライスを置換できない

辞書 (1/2)

```
>>> a = {'program': 'python', 'version': (2,4,3),
'author': 'Guido'}
>>> a['author']           # キーに対応する値を返す
'Guido'
>>> a['licence'] = 'PSL'  # 代入で対応付けをセット
>>> a.has_key('version') # キーに対応する値があるか
True                     # 調べる
>>> a.items()            # 要素の組み合わせを返す
[('program', 'python'), ('version', (2, 4, 3)),
 ('licence', 'PSL'), ('author', 'Guido')]
>>>
```

辞書 (2/2)

- get, setdefault, pop を使うと便利:

```
spec_dict_a = {'cpu': 'Intel', 'disk': 'Seagate',  
'keyboard': 'PFU', 'os': 'FreeBSD', ... }
```

```
spec_dict_b = {'cpu': 'AMD', 'disk':  
'Toshiba', 'Keyboard': 'Kinesys'}
```

'FreeBSD' を返す

```
spec_dict_a.get('os', 'Windows Vista')
```

'Windows Vista' を返す

```
spec_dict_b.get('os', 'Windows Vista')
```

```
spec_dict_b.setdefault('os', 'Windows Vista')
```

```
spec_dict_a.pop('os')
```

'os' の値がないので
'Windows Vista' をセット

'FreeBSD' を返して
辞書から削除

プログラムの構成

`import sys, random` import 文

関数定義

```
def n_dice(faces, trials=1):  
    """<faces>面のサイコロ<trials>回の目を計算する"""  
    nd_sum = 0  
    for i in range(trials):  
        nd_sum += random.choice(range(faces))  
    return nd_sum
```

main ブロック

```
if __name__ == '__main__':  
    ↔ if len(sys.argv) == 3:  
        インデント fc, tr = sys.argv[1:3]  
        ↔ print n_dice(fc, tr)  
    else:  
        print "使い方: ndice.py 面の数 試行数"
```

制御 (1/2)

```
import sys, random
```

```
def n_dice(faces, trials=1):  
    """<faces>面のサイコロ<trials>回の目を計算する"""  
    nd_sum = 0  
    for i in range(trials):  
        nd_sum += random.choice(range(faces))  
    return nd_sum
```

ループ

```
if __name__ == 'main':  
    if len(sys.argv) == 3:  
        fc, tr = sys.argv[1:3]  
        print n_dice(fc, tr)  
    else:  
        print "使い方: ndice.py 面の数 試行数"
```

条件分岐

制御 (2/2)

見方を変えると...

真偽値で制御: if, while
if condition: # 条件分岐

...
while condition: # ループ

- condition の真偽判定
 - 偽: False, 0, [], (), {}, '', None
 - 真: それ以外

iterable で制御: for
for item in iterable: # ループ

- iterable
 - リスト, タプル, 辞書, 文字列
 - iterable オブジェクト

関数

```
import sys, random
    関数名      引数
def n_dice(faces, trials=1):
    """<faces>面のサイコロ<trials>回の目を計算する"""
    nd_sum = 0
    for i in range(trials):
        nd_sum += random.choice(range(faces))
    return nd_sum 戻り値

if __name__ == '__main__':
    if len(sys.argv) == 3:
        fc, tr = sys.argv[1:3]
        print n_dice(fc, tr) 関数呼び出し
    else:
        print "使い方: ndice.py 面の数 試行数"
```

クラスとオブジェクト

```
base_prices = {"Plain":1000, "Cheese":1100, ...}  
topping_prices = {"Spam":150, "Egg":50, ...}
```

```
class Pizza:  
    def __init__(self, base, toppings=()):  
        self.base = base  
        self.toppings = toppings  
    def price(self):  
        p = base_prices[self.base]  
        return p+sum((topping_prices[t]  
                    for t in self.toppings))
```

```
>>> p1 = Pizza("Plain", toppings=["Ham","Egg"])  
>>> p2 = Pizza("Cheese", toppings=[])  
>>> print p1.price()  
>>> 1150
```

モジュールと import (1/2)

- モジュール
 - 関数や変数, クラスなどの寄せ集め
 - 関数名や変数名はモジュールの中で一意
 - プログラムは全て何かのモジュールの一部
- import
 - 他のモジュールの関数名や変数名にアクセス可能にする (自モジュールに取り込む)
 - モジュール名を取り込む
 - モジュール内の関数や変数を取り込む

モジュールと import (2/2)

```
>>> import random          # モジュール名を取り込む
>>> random.choice(range(10)) # モジュール名でアクセス
9
>>> from math import pi    # モジュールの変数を取り込む
>>> math                    # モジュール名は取り込んでない
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'math' is not defined
>>> pi
3.1415926535897931
>>> sin(0.5*pi)           # 取り込んでいない関数は使えない
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'sin' is not defined
>>> from math import *    # モジュールの全名前を取り込む
>>> sin(0.5*pi)
1.0
```

iterable (1/2)

- イテレーション可能って？
 - 辞書では...

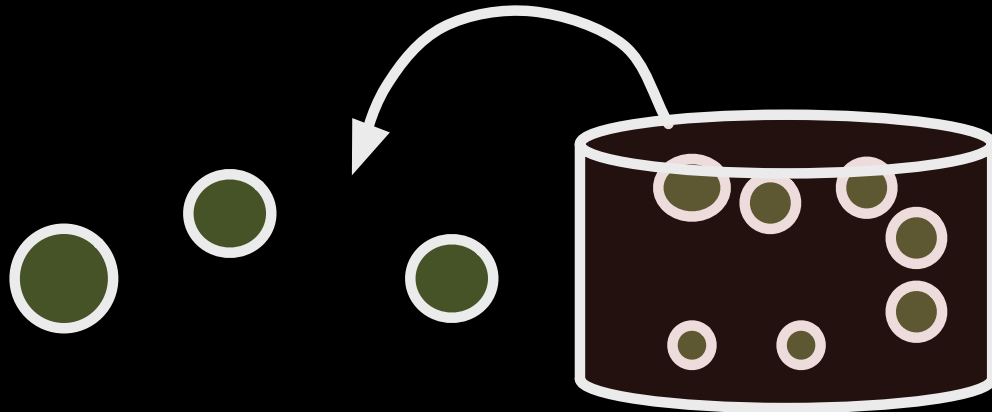
it·er a tion /ɪtə'reɪʃ(ə)n/ → [P]

—n 繰返し, 反復(されるもの); 【数】 反復法
《逐次近似の方法》; 【電算】 繰返し, イテレーション
《終了条件に達するまで一定の処理を繰返すこと》.

(研究社リーダーズ英和辞典)

iterable (2/2)

- Pythonでは？
 - 繰り返し値を取り出せるもの
 - 要するに: `for a in b:` の b に使えるもの
 - リスト, タプル
 - ジェネレータ* etc.



内包表記 (1/2)

[exp for item in iterable]

- コンパクトに書ける
- on-the-fly でリストを作成
- iterable
 - (ただし, リスト全体を一気に作る)
- 入れ子にできる

内包表記 (2/2)

使用前:

```
central_league = ['Tigers', 'Dragons', ...]
pacific_league = ['Buffalos', 'Hawks', ...]
match_results = []
for ce in central_league:
    row = []
    for pa in pacific_league:
        row.append(match_result(ce, pa))
    match_results.append(row)
```

使用後:

```
match_results = [[match_result(ce, pa)
                  for pa in ['Buffalos', 'Hawks', ...]]
                 for ce in ['Tigers', 'Dragons', ...]]
```

ジェネレータ (1/3)

- Generator って？
 - 値を作って (Generate) 返す機能 (function)
 - 見かけは関数
 - iterable (Generator object) を返す.
- 関数名で呼び出されるのは **1回だけ**.
 - 値を取り出すたびに、関数で定義した処理
 - 無限に値を作ってもよい

ジェネレータ (2/3)

```
def match_result_generator():
    central_league = ['Tigers', 'Dragons', ...]
    pacific_league = ['Buffalos', 'Hawks', ...]
    games = 0
    for ce in central_league:
        for pa in pacific_league:
            yield 'Game %d: %s vs %s' %(games, ce, pa)
            games += 1
# pick up object one-by-one from a generator
mr_gen = match_result_generator()
print mr_gen.next()
print mr_gen.next()
...

# usable in for-loop
for line in match_result_generator():
    print line
```

ループ部分

ジェネレータ (3/3)

内包とジェネレータの違い

- 値を取り出すまで計算しない。
データをためない。
- 無限に値を取り出せるジェネレータ

```
from random import sample
def endless_match_generator():
    teams = ['Tigers', 'Dragons', 'Buffalos', 'Hawks', ...]
    game_count = 0
    while 1:
        team_a, team_b = sample(teams, 2)
        yield 'Game %d: %s vs %s' %(game_count, team_a, team_b)
        game_count += 1

for line in endless_match_generator(): # endless!
    print line
```

ジェネレータ式

- (`exp for item in iterable`)
- 内包表記的にジェネレータを書ける
 - 値を作っては返すので効率的
 - コード量が少ない: Don't Repeat Yourself!
 - 関数より `iterable` に見える
 - 間違っって無限ジェネレータをつくりにくい

```
teams = ['Tigers', 'Dragons', 'Buffalos', 'Hawks', ...]
match_lines_gen = ((ce, pa,) for ce in ['Tigers', 'Dragons', ...]
                  for pa in ['Buffalos', 'Hawks', ...])
for line in match_lines_gen:
    print line
```

標準ライブラリ

標準ライブラリ

- 正規表現
- Web 関連
 - CGI, HTTPServer, XMLRPC etc.
- GUI

正規表現: re (1/4)

- 正規表現はテキスト処理の「**運転免許**」
 - パターンを書く
 - パターンをコンパイルする→パターンオブジェクト
 - パターンオブジェクトに文字列を入力
→マッチした部分を返す



正規表現 (2/4)

例題: リンクとテキストを抜き出す

```
html_fragment = ""<a href="link1.html">リンクその1</a>  
  <a name="foo" style="color:black" href="link2.html">リンク  
  その2</a>  
  <a name="anchor_1"></a>  
  <a HREF="link3.html">リンクその  
  3  
</A> ""
```

- どこで改行が入るかわからない
- href パラメタがないかもしれない
- 大文字と小文字のタグ混在

正規表現 (3/4)

- パターンを書く
 - 「”<a”の後で, ”>”より前にある, ”>”を含まない複数の文字」
 - 「”<a ...>”の後で, ”>”より前にある, ”>”を含まない複数の文字」

```
<a([>]+)>([<]+)</a>
```

丸括弧を使うと、一致した部分を
後で取り出しやすい

正規表現 (4/4)

```
>>> import re
>>> ptn_1 = re.compile(r'<a ([^>]*)>([^<]*)</a>',
re.I|re.S|re.M)
>>> ptn_1.findall(html_fragment)
[('href="link1.html"', '\xa5\xea\xa5\xf3\xa5\xaf
\xa4\xbd\xa4\xce1'), ('name="foo" style...), ...]
```

(?name pattern) で, pattern に一致する値を
辞書として取れる

```
ptn_2 = re.compile(r'(?P<key>w+)=(?P<value>w+)',
re.I|re.M|re.S)
for found_1 in ptn_1.finditer(html_fragment):
    aparam, atext = found_1.groups()
    for found_2 in ptn_2.finditer(aparam):
        aurl = m.groupdict.get('href', None)
        if aurl!=None:
            print "%s : %s" %(aurl, atext)
```

Webアプリケーション開発

「要らないよ！僕たちには



django

があるんだ！」



という声もあるでしょうが...

CGI (1/3)

基本は `cgi`, `cgitb` モジュールで十分

```
#!/usr/bin/env python
```

```
import sys, cgi, cgitb
from string import Template
```

```
tmpl = Template("Content-type: text/html\n\n" \
                "<html><body>$content</body></html>")
```

```
sys.stdout.write(tmpl.substitute({'content': 'Hello\nworld!'}))
```



CGI (2/3)

フォームからデータを取り出す

```
import cgi
```

```
form = cgi.FieldStorage()
```

```
# 残念ながら get() は使えない
```

```
if (form.has_key("name") and form.has_key("addr")):
```

```
    print "<p>お名前:", form["name"].value
```

```
    print "<p>ご住所:", form["addr"].value
```

```
if (form.has_key("phone")):
```

```
    for pn in form.getlist("phone"): # 複数値の場合
```

```
        print "<p>お電話:", pn
```

CGI (3/3)

エラーの起きたスクリプトのトレースバックを表示する

```
import cgi
cgi.enable()
```

```
Mozilla Firefox
http://localhost/test.cgi
はじめよう 最新ニュース PyDocJP Search Gmail - 受信トレイ ophidian ymasuda.jp Django Documents Google グループ : turb... Planet P
Hello world!
ValueError
A problem occurred in a Python script. Here is the sequence of function calls leading up to the error, in the order they occurred.
/Users/yasuda/Sites/cubelab/test.cgi
6 cgi.enable()
7 tmpl = Template("Content-type: text/html\n\n\n" \
8                 "<html><body>$content</body></html>")
9 sys.stdout.write(tmpl.substitute({'content': 'Hello world!'}))
10 raise ValueError
builtin ValueError = <class exceptions.ValueError at 0x7d20>
ValueError:
  args = ()
```

*HttpServer

- pure Python の HTTP サーバ
 - BaseHTTPServer: サーバ開発用のベース
 - SimpleHTTPServer: 静的ファイルサーバ
 - CGIHTTPServer: CGI 機能つきサーバ

SimpleHTTPServer

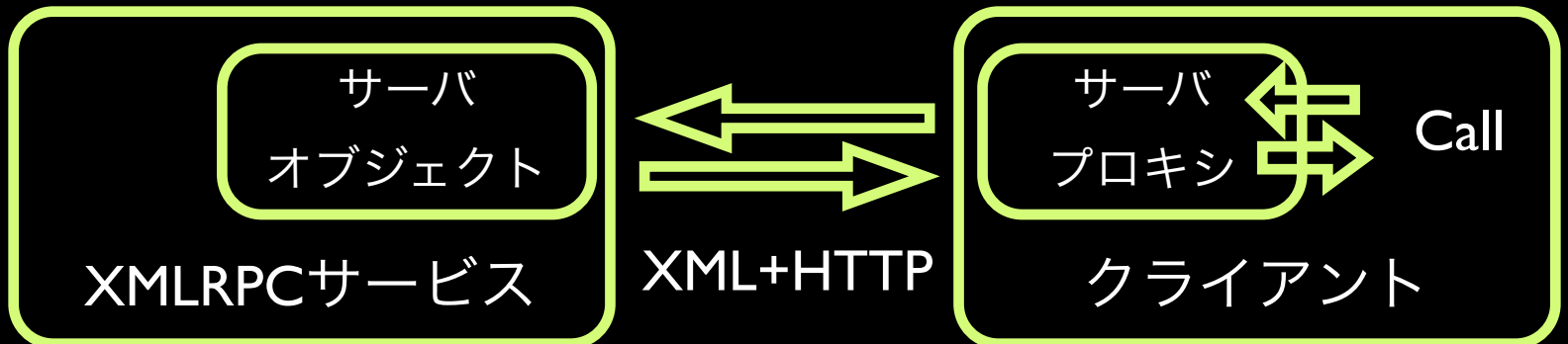
```
from BaseHTTPServer import HTTPServer
from SimpleHTTPServer import SimpleHTTPRequestHandler

# サーバのホスト名とアドレス
server_address = ('localhost', 8000)
# サーバインスタンスを作成
server_instance = HTTPServer(server_address,
                             SimpleHTTPRequestHandler)
# 起動
server_instance.serve_forever()
```

XMLRPC (1/3)

- RPC: Remote Procedure Call
 - (リモート) サーバ上の機能呼び出す
- XMLRPC
 - 呼び出し/戻り値処理の手続きで、データをXMLでコードし、HTTPで送受信

Socket よりお手軽, 高水準, クリーン



XMLRPC (2/3)

- サーバ機能: SimpleXMLRPCServer
 - (リモート) サーバ上の機能呼び出す
- クライアント機能: xmlrpcclib
 - 呼び出し/戻り値処理の手続きで, データをXMLでコードし, HTTPで送受信

XML-RPC (3/3)

サーバ側

```
def myfunc(a, b):  
    return a+b
```

```
from SimpleXMLRPCServer import SimpleXMLRPCServer  
server = SimpleXMLRPCServer(("localhost", 8000))  
server.register_function(myfunc)  
server.serve_forever()
```

クライアント側

```
from xmlrpclib import ServerProxy  
server = ServerProxy("http://localhost:8000/")  
print server.myfunc(a, b)
```

Tkinter (1/3)

- ほとんどの Python 環境で使える GUI ツールキット
 - ウィジェット：ウィンドウやボタン
 - パック：ウィジェットを入れ子にすること
 - コールバック：ウィジェット操作時に呼び出される関数

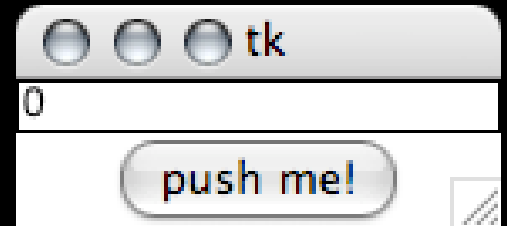
Tkinter (2/3)

ウィジェットの作成とパック

```
import Tkinter
# ルートウィジェットの作成
root = Tkinter.Tk()
# テキスト入力の作成と初期値設定
counter = Tkinter.Entry(root)
counter.insert(0, "0")
# ボタン作成
button = Tkinter.Button(root, text="push me!")

# パック
counter.pack(side='top') # 上詰め
button.pack(side='top') # 上詰め

root.update() # ウィンドウサイズ自動調整
root.mainloop() # 表示
```



Tkinter (3/3)

イベントの設定

```
...
def counter_increment(event):
    value = int(counter.get())           # 値を取り出す
    counter.forget()                    # 消去
    counter.insert(0, str(value+1))     # +1 値を設定
...
button.bind("<Button-1>", counter_increment)
```

その他のトピック

その他のトピック

- 日本語の扱い
- パッケージング

日本語文字列

日本語を使うと...

```
>>> print "あいう"          # 通常の文字列だと OK?
あいう
>>> print u"あいう"         # unicode 文字列では ?
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeEncodeError: 'ascii' codec can't encode
characters in position 0-5: ordinal not in range(128)
```

環境によっては...

```
>>> print "あいう"          # 「文字化け」
$$$$&
```

日本語の扱い

日本語にまつわる問題

- デコードの問題 (string→unicode)

UnicodeDecodeError: 'ascii' codec can't decode byte 0xa4 in position 0: ordinal not in range(128)

- エンコードの問題

UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-6: ordinal not in range (128)

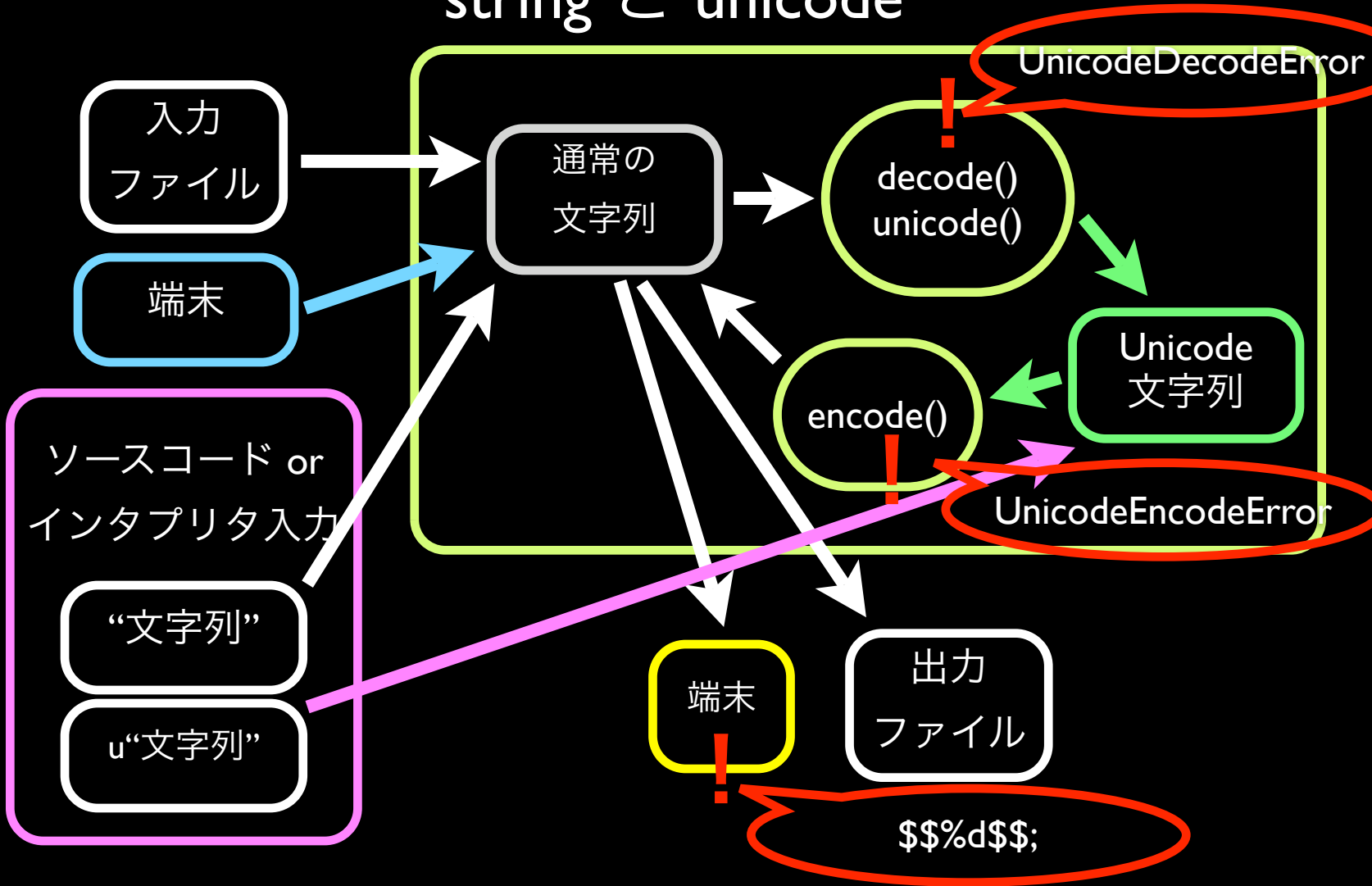
- 文字化け

'\$"\$\$\$&', 'α|α;α α|α;α|α '

文字列と文字コード

- 日本語文字列を表現できる文字コード
 - EUC-JP (拡張UNIX), シフトJIS, JIS etc.
 - UTF-8 etc ← Python の Unicode とは違います
- Python 内部での日本語文字列表現
 - unicode 文字列を使わない場合, Python にとってデータはただのバイト列
 - unicode を使う場合, 入出力時に string への変換が必要

string と unicode



日本語ではまらないために

- 明示的にエンコード／デコードする
 - LANGに依存しなくてもちゃんと動く
 - 内部は全てunicodeデータなので、効率悪い
 - 入出力文字コード判別はプログラマの責任
- 正しくLANG環境を整える
 - デフォルト文字コードが自動設定される
 - 入出力の文字コードを気にしなくてよい
 - LANG をちゃんと設定しない OS が沢山
 - LANG の設定に従わない端末もちらほら

パッケージング: distutils

- パッケージ管理のための 7 徳ツール
- 実際に使うのは 4 徳くらい
 - モジュール配布物のインストール
 - モジュール配布物の作成
 - グローバルパッケージインデクスへの登録
 - スタンドアロンパッケージの作成

distutils

- 手順は簡単
 - ファイルを整理する
 - `setup.py` を作る
 - ソース配布物ビルド
`python setup.py sdist`
 - バイナリ配布物ビルド
`python setup.py bdist (bdist_rpm, bdist_wininst)`
 - PyPI デビュー
`python setup.py register`

setup.py

```
from distutils.core import setup

setup(
    name = 'pyNantoka',
    maintainer = 'Yasushi Masuda',
    maintainer_email = 'ymasuda@example.com',
    description = 'Python-based Nantoka',
    packages = ['nantoka', 'nantoka.hoge',
               'nantoka.moge'],
)
```

ファイル群

```
.../nantoka/__init__.py
.../nantoka/hoge.py
.../nantoka/moge.py
```

さいごに

- Python 標準ライブラリ
 - 面白いモジュールが沢山
 - 色んなプログラミングパラダイムに会える
 - 今も変化し続けている
- ライブラリドキュメント
 - <http://www.python.jp/doc/>
 - 是非一読を
 - 気づいた点があればご連絡ください
 - お手伝い，貢献はいつでも大歓迎です