スクリプト言語プログラミング
Pythonによる数値解析

第７回講義資料

箕原辰夫

オブジェクトのリストの要素

- オブジェクトのリストの確保
  - それだけで、オブジェクトが生成される訳でない
- オブジェクトのリストの要素
  - 繰返しなどを使って、オブジェクトを生成する必要がある
  - 例：

```
complist : [ complex ] = [ None ] * 10
for i in range( len( complist ) ):
    complist[ i ] = complex( i + 3j )
    # 要素のオブジェクトを作る
```

リスト内の要素オブジェクトのフィールド・メソッド

- インデックスを使ったアクセスの方法
  - 属性あるいはフィールド（変数）をアクセスするとき
    - オブジェクトリスト[インデックス].フィールド名
    - complist[5].real
  - オブジェクトのメソッド（関数）を呼び出すとき
    - オブジェクトリスト[インデックス].メソッド名(実パラメータ)
    - complist[i+1].conjugate( )

文字列用のメソッド

- 比較
- 検索
- 置換
- その他
- リストとの変換
- 正規文字列を使った検索

文字列の比較、照合など

- **文字列は、==で等しいかどうか比べる**
- **len**(文字列)…文字列の長さ
- 検索文字列 **in** 検索対象文字列 …入っているかどうか
- **startswith**(比べる文字列)…その文字列で始まるか
- **endswith**(比べる文字列)…その文字列で終るか
- 文字列 不等号演算子 比べる文字列 …辞書順に比べる
  - < 辞書順（Unicode順）で先になっていればTrue
  - > 辞書順（Unicode順）で後になっていればTrue
  - >=, <=, !=なども用いることができる

文字列の検索

- **find**(検索文字列)…文字列の最初から検索、
  返される位置は0から、見つからなければ-1
- **find**(検索文字列, 開始位置)…開始位置から検索
- **find**(検索文字列, 開始位置, 終了位置)…終了位置の手前まで検索
- **index**(位置 [, 開始位置 [, 終了位置]]) …findと同じだが、見つからなければValueError例外扱いになる
- **rfind**(位置 [, 開始位置 [,終了位置]])…findと同じだが、文字列の最後から検索
- **rindex**(位置 [, 開始位置 [, 終了位置]])…indexと同じだが、文字列の最後から検索
- **count**(検索文字列 [, 開始位置 [,終了位置]])…検索文字列が出現する回数を返す

文字列の置換

- 文字列の足し算とスライスを使う
  - s = "This is a sample."
  - result = s[0 : 8] + "the" + s[9 :]
- **文字列は、変更不可能（immutable）なので、新しい文字列を生成するしかない。**
- **replace**(元の文字列, 置換文字列)…該当する文字列を置換する
  なお、３番目のパラメータで置換の最大回数を指定できる
- 置換された新しい文字列が返される

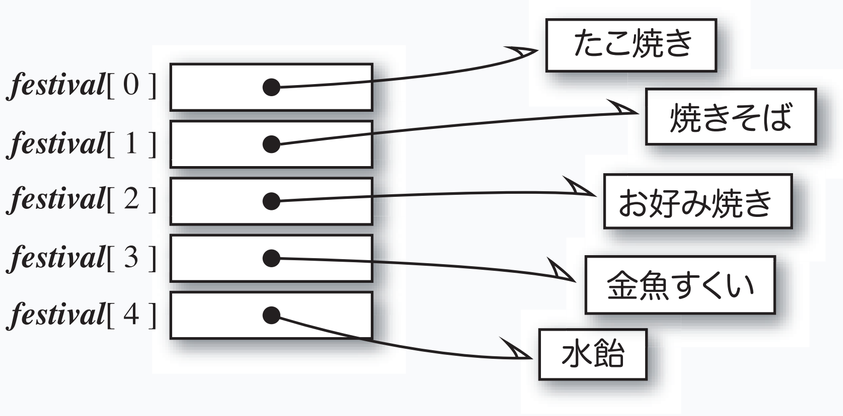
```
s = "This is a sample message for you."
result = s.replace( "a", "the" )
result2 = s.replace( "e", "é", 2 ) # 2回までに制限
```

その他のstringの関数

- strip( ), lstrip( ), rstrip( )…不要な空白を取り除く
- ljust(文字数), rjust(文字数)…文字数分だけの文字列を確保、左右に寄せる、埋める文字は空白が標準だが、２番目のパラメータを与えることによって指定できる
- casefold( )…大文字小文字の区別をしないで等価性を判定する文字列を返す
- upper( ), lower( )…大文字にする、小文字にする
- capitalize( ), title( )…最初の１文字を大文字にして、後の文字を小文字にする、titleは、単語ごとにそれを行なう
- partition(区切り文字列)…文字列を区切り文字列の最初の出現位置で区切り、(区切りの前の部分, 区切り文字列, 区切りの後ろの部分)の３要素から構成されるタプルを返す
- zfill(文字数)…足りなければ、文字数を充たすだけ0を入れる

文字列リストの場合

- 文字列のリスト名[インデックス].メソッド名(実パラメータ)
- 文字列のリスト名[インデックス]スライス



- festival[2].find("お好み")
- festival[2][2: 5]

- リストのスライスと文字列のスライスの組合せは機能しない
- × festival[2: 4][2: 5] # スライスが2回施されたリストとされ、空のリストが返される

文字列とリストの変換

- list(文字列)… 1 文字ずつ分解したリストができる
- str(リスト)…print関数で表示されるような文字列ができる

- 文字列.split(*sep*=区切り文字)…文字列を区切り文字で区切って、リストにする
  区切り文字を指定しないと、タブ（複数も可能）、空白（複数も可能）、改行（複数も可能）で区切ってくれる
- 文字列.splitlines( )…文字列を改行文字で区切って、リストにする
- 区切り文字.join(リスト)…リストの各要素を区切り文字で繋げて文字列にする、ただし、リストは要素が文字列のリストでなければならない

文字列の分割

- stringクラス用のsplitメソッド
- １つの文字列を、区切りとなる文字列で分割し、文字列のリストに直すことができる
- 文字列.split(区切りとなる文字列)
  - 例：

```
tokens = "A sample message".split( " " )
# tokens = [ "A", "sample", "message" ]
for token in tokens:
    print( token )
```

文字列の統合

- 文字列のリストから、間にいれる文字列を指定して、１つの文字列に統合することができる
- "間の文字列".join(文字列のリスト)
  - 例：

```
colorlist = [ "red", "blue", "green", "white" ]
total = " and ".join( colorlist )
# total = "red and blue and green and white"
print( total )
```

正規文字列

- UNIXの伝統の正規文字列
- .　…　任意の一文字
- ?　…　任意の0文字か 1 文字
- *　… 0回以上の繰返し
- + … 1回以上の繰返し
- \a　… その文字自体を表わす　\.　\?　\\　*
- [abc]　…　文字のグループ化　例：[a-z,A-Z,0-9]　[a-z0-9]　[b-x]
- [^abc]　… ^はそれは含めない
- ^ …行の先頭
- $ …行の最後

正規文字列を使った検索

- 正規文字列を扱うモジュール re
  - **import** re # 利用する前に
- 正規文字列で検索
  - マッチオブジェクト = re.search(正規文字列, 検索対象)
  - 例：matset = re.search("[0,4-9]+th", "the 45th event")
- マッチオブジェクトを用いて検索結果
  - 見つからなければNoneになっている
  - start(), end()で、見つかった先頭の位置、終了の位置を求められる
- re.findall(正規文字列, 検索対象) で検索された文字列のリストを得ることができる
  - 例: numlist = re.findall("[0-9]+", "the 123 students get 623 texts.")

Set

- 集合型は、{ }で囲むか、set関数を使ってリストなどから生成する、**for**文を使っても生成することができる
  - 例：fruits = { "りんご", "みかん", "なし", "いちご" }
  - 例：xset = { x**2 **for** x **in** range(1, 11) **if** x % 2 == 1 }
  - 例：drinks = set(["紅茶", "コーヒー" , "ジュース"])
- 集合なので、重複項目は、除去される
- **in** / **not in** 演算子を使って、集合に入っているかどうかを判定
  - 例："なし" **in** fruits
- 組込み関数lenで、集合内の要素の個数を調べることができる

Setと要素

- 集合.add(要素) で、要素を追加する
- 集合.remove(要素)で、要素を削除する、ただし要素がないとエラー
- 集合.discard(要素)で、含まれていればその要素を削除する
- 集合.pop( )で、任意の要素を返し、その要素を削除する、ただし 1 つも要素がないとエラー
- 集合.clear( )で、集合の要素を全部削除する

Setでの演算

- 集合同士の演算→結果の集合が新たに生成されて返される
  - 和集合 union　A | B
  - 差集合 difference　A - B
  - 交差集合 intersection A & B
  - 排他的論理和集合 symmetric_difference　A ^ B = (A | B) - (A & B)
- 部分集合の判定→論理値が返される
  - 部分集合(subset）か　A <= B
  - 真部分集合か　A < B
  - 上位集合（superset）か　A >= B
  - 真上位集合か A > B

辞書（Dictionary）

- 辞書では、キーでエントリの値を引いてくることができる、ただしキーは一意である必要がある
- キー値：対応する値のペア（エントリ: entryと呼ばれる）を、{ }の中に入れることで、辞書を作ることができる。for文を利用して作成することも可能
- 例：flowers = { "rose" : "薔薇", "lily": "百合", "hydrangea": "紫陽花" }
- 例： numbers = { n+1: name **for** n, name **in** enumerate(["one", "two", "three"]) }
- dict関数を使っても、タプルを要素としてもつリストなどから辞書を作成することができる
- 例：numbers = dict(zip(["one", "two", "three"], range(1, 4)))

辞書とエントリ

- 辞書[key]…keyの値を持つvalueを返す
- 辞書[key] = value…keyとvalueの組み合わせを登録
- **del** 辞書[key]…指定されたキーのエントリを削除する
- 辞書.clear( )…全キー（エントリ）をクリアする
- key **in** 辞書 / key **not in** 辞書…keyが辞書にあるか／ないかどうか論理値で返す

辞書の一覧

- len( )…エントリの個数を返す
- keys( )…辞書にあるすべてのキーを返す
- values( )…辞書にあるすべての値を返す
- items( )…辞書にあるすべてのエントリを返す

Iterator

- 何か要素が入っている構造物（コンテナ：containerと呼ばれる）があるときに、繰返しで要素を取り出せるようにするイテレータ型が用意されている
- iter(構造物)関数を用いる
- 例：drinks = { "紅茶", "コーヒー" , "ジュース" }

  for n **in** iter(drinks) : print(n) # 要素を表示
- 例：numbers = dict(zip(["壱", "弐", "参"], range(1, 4)))

  for n **in** iter(numbers) : print(n) #キーを表示
- 辞書の場合には、iterの対象は、キーになる

Iteratorクラスの組込み関数

- iter(コンテナ型のオブジェクト)…Iteratorクラスにする
- next(イテレータ)…次の要素を返す
- 使用例：通常は前のスライドのようなfor文で使う

```
イテレータ = iter( 要素を持つオブジェクト)
while True:
    try:
        要素 = next(イテレータ)
    except StopIteration:
        break
    print(要素)
```

構造体としてのクラス

- クラスの定義
  - **class** クラス名: **pass**
  - 例： **class** Employee: **pass**
- クラス名は、通常、大文字始まりで、複合語であれば、そこを大文字にする命名が多い
（Javaの流儀：Pascal Case / Upper Camel Case）
    - 例：BinaryOperator, SquareMatrix, RawTurtleなど
- Python2からのものや、準原始型に対応するものは、すべてlowercaseにしている場合もある、例：date, datetimeなど

- クラスに属するオブジェクトの生成
  - クラス名( )
  - 例：Employee( )

- 生成したオブジェクトを参照する変数に代入
  - 変数名 = クラス名( )
  - 例：john = Employee( )

構造体としてのクラス

- あるクラスに属するオブジェクトのことをインスタンス(Instance)と呼ぶ。オブジェクトは、独自に変数をもつことができる。これをインスタンス変数と呼ぶ

- インスタンス変数を定義する
  - オブジェクト変数.変数名 = 値の代入
  - 例： john.name = "John Smith"

- 代入することによって、そのオブジェクトに属するインスタンス変数ができる

一般的なクラスの定義

- インスタンス変数（インスタンス属性）とメソッドを持つクラス

```
class クラス名:
    メソッドの定義
```

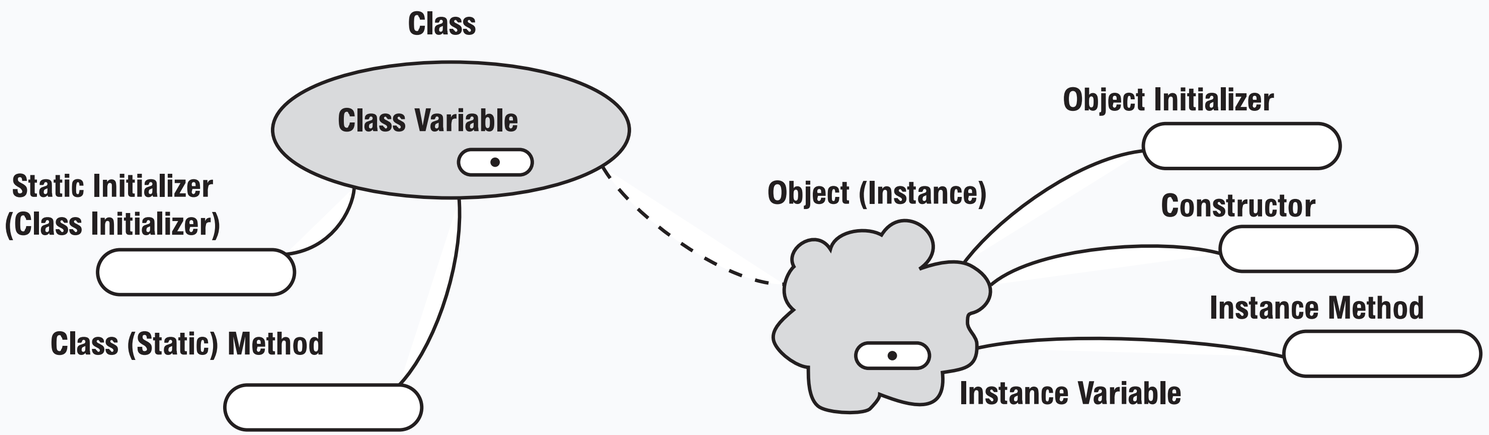
- メソッドを含む、クラスの定義の例：

```
class Binary:
    #インスタンス変数としてself.valueを用いる

    def setValue( self, n ): self.value = ( n != 0 )
    def getValue( self ): return 1 if self.value else 0
```

クラスとオブジェクトの関係

- クラス–オブジェクト関係図



総合的なクラスの作成

- メソッド
  - コンストラクタ・デストラクタ（**self**引数あり）
  - クラスメソッド（**self**引数なし）
    - @classmethodによるもの（cls引数あり）
    - @staticmethodによるもの（cls引数なし）
  - インスタンスメソッド（**self**引数あり）

- 変数
  - クラス属性（**self**なし）
  - インスタンス属性（**self**つき）

コンストラクタ

- オブジェクト生成時に呼び出される特殊メソッド
  - __init__（ **self**, パラメータ名)

- 通常はクラス名( )のコンストラクタは仮定されている。もし、クラス名（　パラメータ名　）のコンストラクタを定義した場合は、上記の仮定が外れる。そのため、__init__（ self ）のコンストラクタを定義する必要がある。

- コンストラクタはインスタンスメソッドなので、**self**が使える。また、スーパークラスのメソッドを呼び出す**super()**も用いることができる

コンストラクタの例

```
class Fraction:
        def __init__( self,  num=0, denom=1 ) :
          self.numerator = num
          self.denominator = denom



number = Fraction( 7, 12 )
another = Fraction( )
```

構造体として属性データをまとめる

- dictを使った場合

```
employee = { "name":"John", "age": 23, "dept":"sales" }
print( employee[ "name" ], employee[ "age" ], employee[ "dept" ] )
```

- クラス定義とコンストラクタを使った場合

```
class Employee:
    def __init__( self, name, age, depart ):
        self.name = name; self.age = age; self.depart = depart

employee = Employee( "John", 23, "sales" )
print( employee.name, employee.age, employee.depart )
```

その他の特殊メソッド

- 特殊メソッドは、言語内部で、様々な場面で呼び出される
  - https://docs.python.org/ja/3.8/reference/datamodel.html
- デストラクタ
  - オブジェクトが削除されるときに呼ばれる
  - **def** __del__(self): で定義
- 文字列化メソッド
  - print関数などで、オブジェクトを文字列に変換されるときに呼ばれる
  - **def** __str__(self): で定義
  - __repr__(), __bytes__(), __format__()も定義することができる
- 演算のためのメソッド
  - 比較演算子 True/Falseを返す

    __eq__, __lt__, __le__, __gt__, __ge__, __ne__ 各比較演算子に該当
  - 四則演算子 結果のオブジェクトを返す

    __add__, __sub__, __matmul__, __mul__, __truediv__, __floordiv__, __pow__, __mod__, __divmod__, __lshift__, __rshift__, __and__, __or__, __xor__ 各演算子に該当

with文

- **with** オブジェクト：　ブロック
- ブロック内で、オブジェクトに対しての命令であることが仮定される
- 例：

```
with cal:
    print( year, month, day )  # cal.year, cal.month, cal.day
```

- **with** オブジェクト **as** 省略名：　ブロック
- ブロック内で、オブジェクトを省略名で参照できる
- 例：

```
with open( "data.text", "r" ) as f:
    print( f.read( ) )
```

with文のためのメソッド

- __enter__(self)と__exit__(self)が用意されていなければならない
- with文に突入するときは、__enter__メソッドが呼ばれる。with文が終了するときは、__exit__メソッドが呼ばれる
- 例：

```
class Tester:
    def __enter__( self ): return self
    def __exit__( self, exc_type, exc_value, traceback ): pass

a = Tester()
a.name = "Kent"
with a as b:
    b.name = "John"
print( a.name )
```

クラス変数（クラス属性）

- そのクラスに（そのクラスに属するオブジェクトに）共通で 1 つの値を保持することができる変数を定義できる
- クラス変数には、**self**修飾子を用いない
- オブジェクトの内部からも、外部からもアクセスするときは、
  - クラス名.クラス変数名
- でアクセスする。

クラス変数の定義

- クラスの中に属性（フィールド）を持つための変数を定義できる。クラス変数と呼ばれている。

```
class クラス名:
    クラス変数の定義
```

- クラス変数は、オブジェクトを生成しなくても、クラス外部からクラス名を使って直接アクセスすることができる。
- 例：

```
class Tester:
    sample = "Hello" #クラス変数の定義
print( Tester.sample )
```

クラス変数での注意点

- クラス定義時において、クラス変数における代入の際に、既出のクラス変数を用いたリスト生成などは利用することができない（クラス定義が関数定義のスコープを利用して実装されているため）
- 例：

```
class Allowed: # 単なる参照はだいじょうぶ
    a = 42
    b = a + 67
class Denied:  # リスト生成において参照されている場合
    a = 42
    b = [ a + n for n in range( 10 ) ] # リスト生成においてクラス変数aを参照
# 後から代入する分にはOK
Denied.b = [ Denied.a + n for n in range( 10 ) ]
```

クラスメソッド

- クラス属性やスタティックメソッド・クラスメソッドは、クラスだけが持つことのできる変数やメソッドである。

- スタティックメソッド・クラスメソッドでは、特定のインスタンスに依拠しているわけではないので、**self**は使えない

- スタティックメソッド・クラスメソッドを呼び出すときは、

  クラス名.スタティックメソッド名(パラメータ)

  という形で呼び出す。

クラスメソッドの種類

- スタティックメソッド
  - @staticmethodデコレータで始まる（つけなくてもOK）
  - 例：

```
class Sample:
    @staticmethod
    def zeroValue(): return 0
```

- クラスメソッド（狭義のPythonのクラスメソッド）
  - @classmethodデコレータで始まる
  - 仮引数の最初が、クラス自身を表すオブジェクトclsになっている
  - 例:

```
class Sample:
    a : int   # class variable
    @classmethod
    def initialize( cls, param):
        cls.a = 10
```

クラスメソッド、クラス変数の例

```
class Color4:
    RED  = "red message"  # クラス変数

    @staticmethod
    def  createBlackColor( ):  # クラスメソッド（引数なし）

        return  "#000000"


    # インスタンスメソッド（コンストラクタ）

    def __init__( self, r, b, g, a ) :
        self.red = r
        self.blue = b
        self.green = g
        self.alpha = a
```

プライベートメンバの指定

- クラス内の属性やメソッドをクラス外から見えなくすることができる。情報の隠蔽化・カプセル化（encapsulation)に用いられる
- 属性の変数、あるいはメソッド名をアンダーバーで始めたものを定義すると、外部から見えなくなる
  - _（アンダーバー１つで始まる）…内部で使うことを意味するが、外部からも参照可能
    - 例：**def** _innerMethod(self, param): **pass**
  - __（アンダーバー２つで始まる）…そのままでは外部からは参照できない。内部では、_<クラス名>__<名前＞に置き換えられているので、それを用いれば参照可能
    - 例：**def** __hiddenMethod(self, param): **pass**

property関数

- オブジェクトで使う変数を隠蔽しておいて、その値を参照する関数（getter）、設定する関数（setter）、削除する関数（deleter）を指定することができる
- 例：

```
class C:
    def __init__(self):  self._x = None
    def getx(self):   return self._x
    def setx(self, value):  self._x = value
    def delx(self):   del self._x
    x = property(getx, setx, delx, "I'm the 'x' property.")
```

- c が C のインスタンスならば、c.x は getter を呼び出し、c.x = value は setter を、del c.x は deleter を呼び出す。
- 最後のパラメータは、doc属性のためのパラメータ

propertyデコレータ

- property組込み関数をデコレータとして使えば、読み出し専用のプロパティを作れる
- 例：

```
class Parrot:
    def __init__(self):  self._voltage = 100000

    @property
    def voltage(self):
        """Get the current voltage."""
        return self._voltage
```

- @property デコレータは voltage() を同じ名前のまま 読み出し専用属性の "getter" にし、voltage のドキュメント文字列を "Get the current voltage." に設定する。
- pがParrotクラスのオブジェクトであれば、p.voltageで値を読み出せる

propertyデコレータ続き

- property オブジェクトは getter, setter, deleter メソッドを持っている。これらのメソッドをデコレータとして使うと、対応するアクセサ関数がデコレートされた関数に設定された、 property のコピーを作成できる。
- 例:

```
class C:
    def __init__(self):  self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):  self._x = value

    @x.deleter
    def x(self):  del self._x
```

継承とクラス

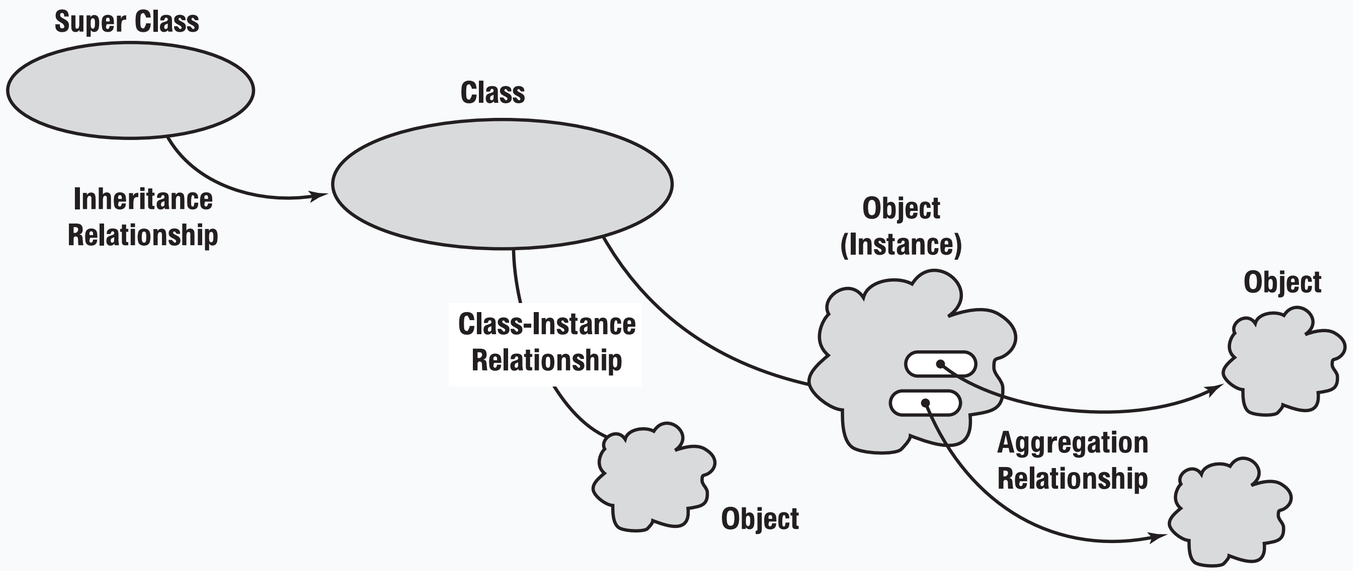
- スーパークラスの変数（クラス変数、インスタンス変数）は、サブクラスから利用ができる

- スーパークラスのメソッド（クラスメソッド、インスタンスメソッド）も、サブクラスから利用ができる

- メソッドについては、同じ名前のメソッドを定義することにより、サブクラスで上書き（overwrite）することが可能になる。

継承関係とクラス–インスタンス関係

- 継承関係図



クラス・オブジェクト間の関係の確認

- オブジェクトの等価性
  - a is b # aとbがオブジェクトとして等しいものであるか（同じidを持っているか）
  - id(a) == id(b) # id組込み関数で、オブジェクトの識別子が返ってくる
- クラスのインスタンス関係の確認
  - isinstance(a, cls) # aがclsのインスタンスとして所属しているか（サブクラスもあり）
  - type(a) == cls # typeは、オブジェクトの所属するクラスを返す
- スーパークラスとサブクラスの関係（継承関係）の確認
  - issubclass(sub_class, super_class) # sub_classがsuper_classのサブクラスになっているか（子孫・祖先なども含む）
  - type(super(sub_class, sub_class)) == super_classはできない（class 'super'になるので）
  - sub_class.__bases__属性に、継承されたクラスがタプルで入っている（多重継承のため）ので、各要素をsuper_classと比較する（ただし内部属性のため、クラス内のメソッドで比較する必要がある
    - 例 **for** bc **in** sub_class.__bases__: print(bc == super_class)

継承と多重継承

- クラス定義の際に、スーパークラスを指定する
- 書式： **class** クラス名(スーパークラス名):
  - 例： **class** Elephant(Animal): **pass**
- スーパークラスは、複数指定することができる（多重継承：multiple inheritance と呼ばれる）
  - 例：**class** SA(Student, Assistant): **pass**
- 多重継承の場合、指定されたスーパクラス名の順番に従って、スーパクラスが優先される

super() 組込み関数

- super()とすると、スーパークラスの変数（属性）やメソッドを利用することができる（クラス内）
- 例：
  - super().method(arg)
    - # スーパークラスのmethodを引数argで呼び出す
    - インスタンスメソッドからの呼出しの場合は、selfが自動でスーパークラスのメソッドに与えられる
- super(クラス名, オブジェクトまたはクラス名)で指定しても呼び出すことができる
- 例：
  - super(Rabbit, self).method(arg)
    - # Rabbitのスーパークラスのインスタンスメソッドmethodを引数argで呼び出す
  - super(Rabbit, Rabbit).clsmethod(arg)
    - # Rabbitのスーパークラスのクラスメソッドclsmethodを引数argで呼び出す

継承を利用した例

- Rabbitクラスは、Animalクラスを継承している

```
class Animal:
    name = "Animal "
    def jump( self ):
        print( "Jump: " + Animal.name )
    def shout( self ):
        print( "WOW:  " + Animal.name )

class Rabbit( Animal ):
    name = "Rabbit"
    def jump( self ):
      print( "Jump: " + Rabbit.name )
      #overwrite
    def superjump( self ):  super().jump( )
```

- 呼出し結果は以下のようになる

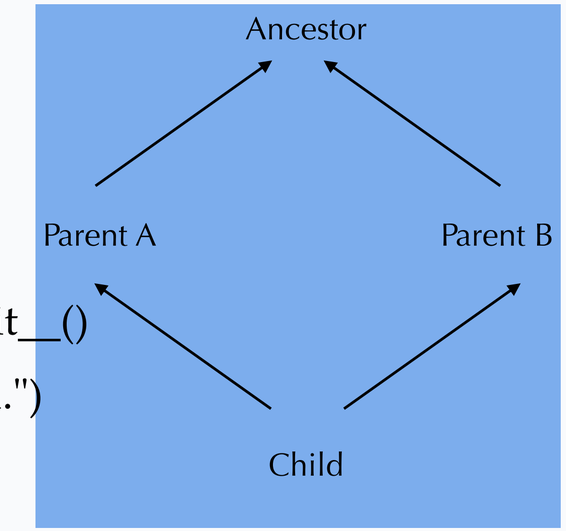
```
Animal().jump()     # Jump: Animalと表示される
r = Rabbit()        # rは、Rabbitクラスのオブジェクト
r.jump()            # Jump: Rabbitと表示される
r.shout()           # WOW: Animalと表示される
r.superjump()       # Jump: Animalと表示される
```

多重継承時の初期化・優先順位

- 右図のような形で多重継承されていた場合、初期化の呼出しはSelf→A→B→Ancester→B→A→Selfという形で起こる
- プログラム例：

```
class Ancestor:
    def __init__(self):  print( "Ancestor initialized.")

class ParentA( Ancestor ):
    def __init__(self):
        print( "Parent A called.")
        super( ParentA, self ).__init__()
        print( "Parent A initialized.")
```




```
class ParentB( Ancestor ):
    def __init__(self):
        print( "Parent B called.")
        super( ParentB, self ).__init__()
        print( "Parent B initialized.")

class Child( ParentA, ParentB ):
    def __init__(self):
        print( "Child called.")
        super( Child, self ).__init__()
        print( "Child initialized.")

me = Child()
```

列挙型

- 値は重要ではなく、区別だけをしたいときには、列挙型を用いる
  - 例：　東西南北　をプログラム中で使いたい
- 列挙型を用いるときは、enumモジュールのEnumを使って、以下のように利用する
- 書式：
  - **from** enum **import** Enum
  - **class** 列挙型のクラス名(Enum):
  - 　変数 = 値
- 例：**from** enum **import** Enum

```
class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3
```

- 列挙型は、代入や比較が可能になっている

例：

```
x = Color.BLUE
return Color.RED == x
```

抽象クラス

- 具体的な記述を持たないスーパークラス
- サブクラスは、メソッドを上書きする必要がある
- Pythonでは、abcモジュールで定義されているABCクラスを継承する、あるいはABCMetaクラスをメタクラスとして指定すると、抽象クラスになる。
- 例：

```
from abc import ABC
class MyABC(ABC):
    pass

from abc import ABCMeta
class MyABC( metaclass=ABCMeta):
    pass
```

抽象クラス使用のメソッド

- @abstractmethodデコレータを使って、上書きされる必要のあるメソッドを外形だけを定義する
- 抽象クラスを継承した（実体のある）クラスでは、外形だけ定義されたメソッドの実体を定義する必要がある
- 例：

```
from abc import ABCMeta
class Animal( metaclass=ABCMeta):
    @abstractmethod
    def eat( self ):
        pass

class Rabbit( Animal ):
    def eat(self):
        print( "Have a lettuce" )
```

モジュールとしての定義

- 1つの.pyファイルに書かれたプログラムは、モジュールと呼ばれる
- 他のPythonのプログラムから、該当のファイルの関数などを呼び出すときは、**import**文を利用して参照することができる。「ファイル名.py」のファイル名の部分がモジュール名として利用される
- 例： matrix.py → **import** matrix などを利用して、中の関数を利用可能
- __name__という内部変数（特殊属性）が、"__main__"という文字列になっていれば、そのモジュールから実行が開始されていることがわかる
- 例： そのモジュールから実行するときだけ、値の入力を要求して、main関数を実行させる

```
if __name__ == "__main__": # このモジュールから実行されたかどうか
    value = input( "値を入力して下さい" )
    main( value ) # そのモジュールから実行されたときだけ呼ぶ
```

  - **import**などで他のモジュールから呼ばれた場合は、この部分は実行されない

ライブラリ（パッケージ・モジュール）を使うとき

- いくつかの記述法があり、プログラム中でどのような名前で利用できるかの違いがある
- **import** [パッケージ名.]モジュール名
  - プログラム中で「モジュール名.名前」で利用可能
  - 例：**import** math;　print(math.pi)
- **import** [パッケージ名.]モジュール名 **as** 省略名
  - 「省略名.名前」で利用可能
  - 例：**import** math **as** mt; print(mt.pi)
- **from** [パッケージ名.]モジュール名 **import** クラスあるいは関数名
  - 「クラス名あるいは関数名」で利用可能
  - **from** math **import** pi; print(pi)
- **from** [パッケージ名.]モジュール名 **import** *
  - モジュールで定義されているすべての「クラス名あるいは関数名」で利用可能
  - **from** math **import** * ; print(e)
- **from** パッケージ名 **import** モジュール名
  - 「モジュール名.名前」で利用可能
  - **from** urllib **import** request; request.urlopen(url)

モジュールの検索

- **import**文で指定されたモジュール（ファイル）がどこにあるかを検索する順番が以下に規定されている

  1.カレントフォルダ

  2.環境変数PYTHONPATHに設定されているフォルダ

  3.標準ライブラリモジュールのフォルダ

  - 例えば、Mac OS XでPython 3.9の場合は、
    /Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/

  4.サードパーティのライブラリのフォルダ

  - 例えば、Mac OS XでPython 3.9の場合は、
    /Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-packages/

- 標準ライブラリのモジュールと同じ名前でファイルをカレントフォルダに作った場合、そちらの方が優先されるので、注意する必要がある。例：math.pyなど

パッケージの利用

- 複数のモジュール（.pyあるいは.pyc）ファイルを1つのフォルダにまとめておき、そのフォルダをパッケージとして利用することができる
- **import**文では、「パッケージ名.モジュール名」などで参照することができる。
  - 例：
    - **from** packageA **import** moduleA
    - **import** packageA.moduleA
- そのフォルダに、「__init__.py」という名前のモジュールがあれば、そのパッケージをimportした際に、最初にこのモジュールが実行されるので、このモジュールに初期化用のプログラムを記述しておく
- パッケージについてもフォルダの検索順序は、モジュールと同様になる