Python入門(3)

- オブジェクト
- クラスとメソッド
- 操作（インスタンス)

Pythonオブジェクト指向

Pythonの仮想空間環境ではすべてのもの（データ）はオブジェクトです。

すべての'データ型'はクラスです。

以前の演習では基本データ型の内容を紹介しました。

- 整数, 浮動小数点数, 複素数, 文字列, リスト, タプル, 辞書, など

In [ ]:
```
type(58) #型
```

Out[ ]:
```
int
```

すべてのオブジェクトには一意のIDがあります。

In [ ]:
```
id(58) #オブジェクト番号
```

Out[ ]:
```
140715040545376
```

- Boolean型

In [1]:
```
help(bool)
```

```
Help on class bool in module builtins:

class bool(int)
 |  bool(x) -> bool
 |
 |  Returns True when the argument x is true, False otherwise.
 |  The builtins True and False are the only two instances of the class bool.
 |  The class bool is a subclass of the class int, and cannot be subclassed.
 |
 |  Method resolution order:
 |      bool
 |      int
 |      object
 |
 |  Methods defined here:
 |
 |  __and__(self, value, /)
 |      Return self&value.
 |
 |  __or__(self, value, /)
 |      Return self|value.
 |
 |  __rand__(self, value, /)
 |      Return value&self.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __ror__(self, value, /)
 |      Return value|self.
 |
 |  __rxor__(self, value, /)
 |      Return value^self.
 |
 |  __str__(self, /)
 |      Return str(self).
 |
 |  __xor__(self, value, /)
 |      Return self^value.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from int:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __bool__(self, /)
 |      self != 0
 |
 |  __ceil__(...)
 |      Ceiling of an Integral returns itself.
 |
 |  __divmod__(self, value, /)
 |      Return divmod(self, value).
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __float__(self, /)
```

```
    float(self)

__floor__(...)
    Flooring an Integral returns itself.

__floordiv__(self, value, /)
    Return self//value.

__format__(self, format_spec, /)
    Default object formatter.

__ge__(self, value, /)
    Return self>=value.

__getattribute__(self, name, /)
    Return getattr(self, name).

__getnewargs__(self, /)

__gt__(self, value, /)
    Return self>value.

__hash__(self, /)
    Return hash(self).

__index__(self, /)
    Return self converted to an integer, if self is suitable for use as an index into a list.

__int__(self, /)
    int(self)

__invert__(self, /)
    ~self

__le__(self, value, /)
    Return self<=value.

__lshift__(self, value, /)
    Return self<<value.

__lt__(self, value, /)
    Return self<value.

__mod__(self, value, /)
    Return self%value.

__mul__(self, value, /)
    Return self*value.

__ne__(self, value, /)
    Return self!=value.

__neg__(self, /)
    -self

__pos__(self, /)
    +self

__pow__(self, value, mod=None, /)
    Return pow(self, value, mod).

__radd__(self, value, /)
    Return value+self.

__rdivmod__(self, value, /)
    Return divmod(value, self).

__rfloordiv__(self, value, /)
    Return value//self.

__rlshift__(self, value, /)
    Return value<<self.

__rmod__(self, value, /)
    Return value%self.

__rmul__(self, value, /)
    Return value*self.

__round__(...)
    Rounding an Integral returns itself.
    Rounding with an ndigits argument also returns an integer.

__rpow__(self, value, mod=None, /)
    Return pow(value, self, mod).

__rrshift__(self, value, /)
    Return value>>self.

__rshift__(self, value, /)
    Return self>>value.

__rsub__(self, value, /)
    Return value-self.

__rtruediv__(self, value, /)
    Return value/self.

__sizeof__(self, /)
    Returns size in memory, in bytes.

__sub__(self, value, /)
    Return self-value.

__truediv__(self, value, /)
    Return self/value.

__trunc__(...)
    Truncating an Integral returns itself.

bit_length(self, /)
    Number of bits necessary to represent self in binary.

    >>> bin(37)
    '0b100101'
    >>> (37).bit_length()
    6
```

```
conjugate(...)
    Returns self, the complex conjugate of any int.

to_bytes(self, /, length, byteorder, *, signed=False)
    Return an array of bytes representing an integer.

    length
      Length of bytes object to use.  An OverflowError is raised if the
      integer is not representable with the given number of bytes.
    byteorder
      The byte order used to represent the integer.  If byteorder is 'big',
      the most significant byte is at the beginning of the byte array.  If
      byteorder is 'little', the most significant byte is at the end of the
      byte array.  To request the native byte order of the host system, use
      `sys.byteorder' as the byte order value.
    signed
      Determines whether two's complement is used to represent the integer.
      If signed is False and a negative integer is given, an OverflowError
      is raised.

----------------------------------------------------------------------
Class methods inherited from int:

from_bytes(bytes, byteorder, *, signed=False) from builtins.type
    Return the integer represented by the given array of bytes.

    bytes
      Holds the array of bytes to convert.  The argument must either
      support the buffer protocol or be an iterable object producing bytes.
      Bytes and bytearray are examples of built-in objects that support the
      buffer protocol.
    byteorder
      The byte order used to represent the integer.  If byteorder is 'big',
      the most significant byte is at the beginning of the byte array.  If
      byteorder is 'little', the most significant byte is at the end of the
      byte array.  To request the native byte order of the host system, use
      `sys.byteorder' as the byte order value.
    signed
      Indicates whether two's complement is used to represent the integer.

----------------------------------------------------------------------
Data descriptors inherited from int:

denominator
    the denominator of a rational number in lowest terms

imag
    the imaginary part of a complex number

numerator
    the numerator of a rational number in lowest terms

real
    the real part of a complex number
```

In [2]:
```
# ブール値
type(False)
```

Out[2]: bool

In [3]:
```
id(False)
```

Out[3]: 94286151377184

In [4]:
```
# 0ではない数字はTrueになる
bool(10)
```

Out[4]: True

In [5]:
```
bool(0.0)
```

Out[5]: False

In [6]:
```
bool(-10)
```

Out[6]: True

In [7]:
```
bool([])
```

Out[7]: False

- キーボードからの入力を待つ

In [18]:
```
a = input("半角数字を入力してください。")
```

半角数字を入力してください。3.14

In [19]:
```
a
```

Out[19]: '3.14'

In [20]:
```
type(a)
```

Out[20]: str

In [21]:
```
b = float(a) #  float型に変換
type(b)
```

Out[21]: float

- 辞書型のオブジェクト

In [24]:
```
d = dict() # 辞書型のオブジェクトdを作成する d={}と同じ
type(d)
```

Out[24]: dict

In [22]:
```
help(dict)
```

```
Help on class dict in module builtins:

class dict(object)
 |  dict() -> new empty dictionary
 |  dict(mapping) -> new dictionary initialized from a mapping object's
 |      (key, value) pairs
 |  dict(iterable) -> new dictionary initialized as if via:
 |      d = {}
 |      for k, v in iterable:
 |          d[k] = v
 |  dict(**kwargs) -> new dictionary initialized with the name=value pairs
 |      in the keyword argument list.  For example:  dict(one=1, two=2)
 |
 |  Methods defined here:
 |
 |  __contains__(self, key, /)
 |      True if the dictionary has the specified key, else False.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __setitem__(self, key, value, /)
 |      Set self[key] to value.
 |
 |  __sizeof__(...)
 |      D.__sizeof__() -> size of D in memory, in bytes
 |
 |  clear(...)
 |      D.clear() -> None.  Remove all items from D.
 |
 |  copy(...)
 |      D.copy() -> a shallow copy of D
 |
 |  get(self, key, default=None, /)
 |      Return the value for key if key is in the dictionary, else default.
 |
 |  items(...)
 |      D.items() -> a set-like object providing a view on D's items
 |
 |  keys(...)
 |      D.keys() -> a set-like object providing a view on D's keys
 |
 |  pop(...)
 |      D.pop(k[,d]) -> v, remove specified key and return the corresponding value.
 |      If key is not found, d is returned if given, otherwise KeyError is raised
 |
 |  popitem(...)
 |      D.popitem() -> (k, v), remove and return some (key, value) pair as a
 |      2-tuple; but raise KeyError if D is empty.
 |
 |  setdefault(self, key, default=None, /)
 |      Insert key with a value of default if key is not in the dictionary.
 |
 |      Return the value for key if key is in the dictionary, else default.
 |
 |  update(...)
```

```
|      D.update([E, ]**F) -> None.  Update D from dict/iterable E and F.
|      If E is present and has a .keys() method, then does:  for k in E: D[k] = E[k]
|      If E is present and lacks a .keys() method, then does:  for k, v in E: D[k] = v
|      In either case, this is followed by: for k in F:  D[k] = F[k]
|
|  values(...)
|      D.values() -> an object providing a view on D's values
|
|  ----------------------------------------------------------------------
|  Class methods defined here:
|
|  fromkeys(iterable, value=None, /) from builtins.type
|      Create a new dictionary with keys from iterable and values set to value.
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  __hash__ = None
```

例えば、辞書型の内容は上記の様になります。

メソッド(methods)とは、仕様書によるそれぞれの操作リストの中の一つです。

知られている属性はデータタプル (キー,値) である。

知られているメソッドはitems(),keys(),values(),pop()などあります。

In [25]:
```
#最初はdの中は空です。次の様に要素を追加する
d['キー']='値'
d
```

Out[25]: {'キー': '値'}

In [26]:
```
d['キー']= 'key' #更新
d
```

Out[26]: {'キー': 'key'}

In [27]:
```
d['値']='value' #追加
d
```

Out[27]: {'キー': 'key', '値': 'value'}

In [28]:
```
d.items()
```

Out[28]: dict_items([('キー', 'key'), ('値', 'value')])

In [29]:
```
d.get('値')
```

Out[29]: 'value'

In [30]:
```
id(d)
```

Out[30]: 140237656052112

In [31]:
```
d['自分']='self'
d['属性']='property'
d['メソッド']='method'
d.items()
```

Out[31]: dict_items([('キー', 'key'), ('値', 'value'), ('自分', 'self'), ('属性', 'property'), ('メソッド', 'method')])

In [32]:
```
k = input('単語を入力：')# キーボードからの入力待ち, '自分'を入力
print('辞書->',k,':',d[k])
```

```
単語を入力：自分
辞書-> 自分 : self
```

- 簡単な操作

問題：　店の服サイズと価格を辞書型に格納され、サイズを与え、価格を出力する。

In [66]:
```
# 辞書
服 = {'こども':None, 'ジュニア':2000,'S':2500,'M':3000,'L':3500,'LL':4000}

def get_price(サイズ):
  if サイズ == 'こども':
    return('価格はありません。')
  else :
    return (服[サイズ])
```

```
print("Sサイズの価格：",get_price('S'),'円')
```

Sサイズの価格： 2500 円

In [67]:
```
print("こどもサイズの価格：",get_price('こども'))
```

こどもサイズの価格： 価格はありません。

In [69]:
```
注文 = ['ジュニア','ジュニア','S','S','S','LL','LL','LL','LL','こども','こども','こども']

注文価格 = list() # 空リストのオブジェクト

for e in 注文:
  注文価格.append(get_price(e)) # リストのappendメソッドを利用し、要素eサイズの価格を追加する

print("注文価格：", 注文価格)
```

注文価格： [2000, 2000, 2500, 2500, 2500, 4000, 4000, 4000, 4000, '価格はありません。', '価格はありません。', '価格はありません。']

オブジェクト指向の理論

クラスとオブジェクト

オブジェクトは、コンピュータにバーチャルに知られているすべての物の記述を提供する独立した項目です。

車、飛行機、自転車などの実世界の物はオブジェクトです。オブジェクトは、データと動作という2つの主要な特性を共有しています。

車には、車輪の数、ドアの数、座席数などのデータがあり、加速、停止、不足している燃料の量の表示などの動作もあります。

オブジェクト指向プログラミングでは、データを属性と呼び、動作をメソッドと呼びます。

また、クラスは、個々のオブジェクトが作成される型です。 現実の世界では、すべて同じタイプのオブジェクトがたくさん見つかります。 車のように。 すべて同じメーカーとモデル（エンジン、ホイール、ドアなど）があります。 各車は同じ設計図のセットから製造され、同じコンポーネントを備えています。

In [35]:
```
# 乗物のクラス定義
class 乗物:
  pass
```

In [36]:
```
#オブジェクトを作成する
乗物1 = 乗物()
print("オブジェクト：",乗物1)
```

オブジェクト： <__main__.乗物 object at 0x7f8b9fb3df10>

In [37]:
```
# 乗物のクラス再定義：座席数の属性を追加する
class 乗物:
  def __init__(self,座席数):
    self._座席数 = 座席数
  @property
  def 座席数(self):
    return self._座席数
```

In [38]:
```
#オブジェクトを作成する
乗物1 = 乗物(10)
print("オブジェクト：",乗物1)
print("乗物1の座席数",乗物1.座席数)
```

オブジェクト： <__main__.乗物 object at 0x7f8b9fb17e50>
乗物1の座席数 10

In [39]:
```
# 継承
# 車クラス定義：乗物クラスの継承,ドア数,エンジン,車輪の数,座席数,メーカーの属性を追加する
class 車(乗物):
  def __init__(self, ドア数, エンジン, 車輪の数, 座席数, メーカー):
    self._ドア数 = ドア数
    self._エンジン = エンジン
    self._車輪の数 = 車輪の数
    self._メーカー = メーカー
    super().__init__(座席数) #親クラスの__init__(座席数)
    #self._座席数 = super().座席数

  @property
  def ドア数(self):
    return self._ドア数
  @property
  def エンジン(self):
    return self._エンジン
  @property
  def 車輪の数(self):
    return self._車輪の数
  @property
  def メーカー(self):
    return self._メーカー
```

```
  def データ出力(self):
    print(self._メーカー, self._ドア数, self._エンジン, self._座席数, self._車輪の数)
```

Wikipediaのリンク: N-BOX

In [40]:
```
# NBOXクラス定義：車クラスの継承, 色,ナンバープレートの属性を追加する
class NBOX(車):
  def __init__(self,ドア数,エンジン,車輪の数,座席数,メーカー,色,ナンバープレート):
    super().__init__(ドア数,エンジン,車輪の数,座席数,メーカー)
    self._色 = 色
    self._ナンバープレート = ナンバープレート

  #property
  def get_color(self):
    return self._色
  #property
  def get_nb(self):
    return self._ナンバープレート
  #set_color.setter
  def set_color(self, 色):
    self._色 = 色
  #set_nb.setter
  def set_nb(self, ナンバープレート):
    self._ナンバープレート = ナンバープレート

  def データ出力(self):
    super().データ出力()
    print(self._色,self._ナンバープレート)
```

In [41]:
```
NBOX1 = NBOX(5,'658cc',4,4,'HONDA','白','岩手500さ00-01')
print("オブジェクトアドレス：",NBOX1)

print("オブジェクトの属性を出力する：")
NBOX1.データ出力()
```

```
オブジェクトアドレス： <__main__.NBOX object at 0x7f8b9fadb410>
オブジェクトの属性を出力する：
HONDA 5 658cc 4 4
白 岩手500さ00-01
```

In [42]:
```
vars(NBOX1)
```

Out[42]:
```
{'_エンジン': '658cc',
 '_ドア数': 5,
 '_ナンバープレート': '岩手500さ00-01',
 '_メーカー': 'HONDA',
 '_座席数': 4,
 '_色': '白',
 '_車輪の数': 4}
```

In [43]:
```
print("NBOX1の座席数:",NBOX1.座席数,'ドア数:',NBOX1.ドア数)
print('エンジン:',NBOX1.エンジン,
      '車輪の数:',NBOX1.車輪の数)
```

```
NBOX1の座席数: 4 ドア数: 5
エンジン: 658cc 車輪の数: 4
```

In [44]:
```
NBOX1._エンジン
```

Out[44]:
```
'658cc'
```

In [45]:
```
NBOX2 = NBOX1
print("オブジェクトアドレス：",NBOX2)
print("オブジェクトの属性を出力する：")
NBOX2.データ出力()
```

```
オブジェクトアドレス： <__main__.NBOX object at 0x7f8b9fadb410>
オブジェクトの属性を出力する：
HONDA 5 658cc 4 4
白 岩手500さ00-01
```

In [46]:
```
NBOX2 is NBOX1
```

Out[46]:
```
True
```

In [47]:
```
NBOX2 == NBOX1
```

Out[47]:
```
True
```

In [48]:
```
from copy import deepcopy
NBOX3 = deepcopy(NBOX1)
NBOX3
```

Out[48]:
```
<__main__.NBOX at 0x7f8b9faf1650>
```

In [49]:
```
print(id(NBOX1), id(NBOX3))
```

```
140237656142864 140237656233552
```

In [50]:
```
NBOX3 == NBOX1
```

Out[50]:
```
False
```

例えば、NBOX1は色とナンバープレートを変わって、NBOX3になるということを考えましょう。 実世界では車自体は変わりないですが、オブジェクト指向の仮想空間ではNBOX1はNBOX3へコピーし、NBOX3の属性を変更し、NBOX1を削除します。削除しなくとも問題ないですが、実世界の内容のようにしたいならば、削除します。

In [51]:
```
NBOX3.データ出力() # 現在のデータ
```

```
HONDA 5 658cc 4 4
白 岩手500さ00-01
```

変更を行う。

In [52]:
```
NBOX3.set_color('青')
NBOX3.set_nb('岩手500さ00-02')
print("オブジェクトの属性を出力する：")
NBOX3.データ出力()
```

```
オブジェクトの属性を出力する：
HONDA 5 658cc 4 4
青 岩手500さ00-02
```

NBOX1オブジェクトを削除する：

In [53]:
```
del NBOX1
```

In [54]:
```
vars(NBOX3)
```

Out[54]:
```
{'_エンジン': '658cc',
 '_ドア数': 5,
 '_ナンバープレート': '岩手500さ00-02',
 '_メーカー': 'HONDA',
 '_座席数': 4,
 '_色': '青',
 '_車輪の数': 4}
```

Wikipedia: 日産・アトラス

In [55]:
```
# クラス定義：車クラスの継承, 色,ナンバープレート,重さの属性を追加する
class アトラス(車):
  def __init__(self,ドア数,エンジン,車輪の数,座席数,メーカー,色,ナンバープレート,重さ):
    super().__init__(ドア数,エンジン,車輪の数,座席数,メーカー)
    self._色 = 色
    self._ナンバープレート = ナンバープレート
    self._重さ = 重さ

  @property
  def get_color(self):
    return self._色
  @property
  def get_nb(self):
    return self._ナンバープレート
  @property
  def get_nb(self):
    return self._重さ

  def データ出力(self):
    super().データ出力()
    print(self._色, self._ナンバープレート)
```

In [56]:
```
t1 = アトラス(2,'Z型',4,3,'NISSAN','白','岩手500さ00-03','1.5t')
print("オブジェクトアドレス：",t1)

print("オブジェクトの属性を出力する：")
t1.データ出力()
```

```
オブジェクトアドレス： <__main__.アトラス object at 0x7f8b9faf1850>
オブジェクトの属性を出力する：
NISSAN 2 Z型 3 4
白 岩手500さ00-03
```

In [57]:
```
print("t1の座席数:",t1.座席数,'ドア数:',t1.ドア数)
print('エンジン:',t1.エンジン,
      '車輪の数:',t1.車輪の数)
```

```
t1の座席数: 3 ドア数: 2
エンジン: Z型 車輪の数: 4
```

In [58]:
```
vars(t1)
```

Out[58]:
```
{'_エンジン': 'Z型',
 '_ドア数': 2,
 '_ナンバープレート': '岩手500さ00-03',
 '_メーカー': 'NISSAN',
 '_座席数': 3,
 '_色': '白',
 '_車輪の数': 4,
 '_重さ': '1.5t'}
```

飛ぶ自動車はどうすればいいですか

飛行機と車の両方を持つ乗物です。

https://en.wikipedia.org/wiki/PAL-V

In [59]:
```
class 飛行機:
  pass
```

In [60]:
```
# 多重継承: 車と飛行機のクラスから継承
class 飛ぶ自動車(車,飛行機):
  pass
```

In [61]:
```
prototype0 = 飛ぶ自動車(1,'電気',3,1,'PAL-V')
type(prototype0)
```

Out[61]: __main__.飛ぶ自動車

In [62]:
```
prototype0.データ出力()
```

```
PAL-V 1 電気 1 3
```

In [63]:
```
vars(prototype0)
```

Out[63]: {'_エンジン': '電気', '_ドア数': 1, '_メーカー': 'PAL-V', '_座席数': 1, '_車輪の数': 3}

In [ ]: