Python for scientific research

Built-in data types

Bram Kuijper

University of Exeter, Penryn Campus, UK

February 11, 2020

Declaring variables

- Variables that contain data (numbers, characters etc.) are the fundamental building blocks of any language

Declaring variables

- Variables that contain data (numbers, characters etc.) are the fundamental building blocks of any language
- Variables in Python are dynamically typed (see later)

Declaring variables

- Variables that contain data (numbers, characters etc.) are the fundamental building blocks of any language
- Variables in Python are dynamically typed (see later)
- You can name your variables pretty much anything you want but:

Declaring variables

- Variables that contain data (numbers, characters etc.) are the fundamental building blocks of any language
- Variables in Python are dynamically typed (see later)
- You can name your variables pretty much anything you want but:
  - Be **consistent** in your naming convention (see PEP 8)

Declaring variables

- Variables that contain data (numbers, characters etc.) are the fundamental building blocks of any language
- Variables in Python are dynamically typed (see later)
- You can name your variables pretty much anything you want but:
  - Be **consistent** in your naming convention (see PEP 8)
  - Python is **case sensitive**; `genename` and `geneName` are different variables

Declaring variables

- Variables that contain data (numbers, characters etc.) are the fundamental building blocks of any language
- Variables in Python are dynamically typed (see later)
- You can name your variables pretty much anything you want but:
  - Be **consistent** in your naming convention (see PEP 8)
  - Python is **case sensitive**; `genename` and `geneName` are different variables
  - First character cannot contain a digit: `1genename` raises an error

Declaring variables

- Variables that contain data (numbers, characters etc.) are the fundamental building blocks of any language
- Variables in Python are dynamically typed (see later)
- You can name your variables pretty much anything you want but:
  - Be **consistent** in your naming convention (see PEP 8)
  - Python is **case sensitive**; `genename` and `geneName` are different variables
  - First character cannot contain a digit: `1genename` raises an error
  - You cannot use Python's **reserved words/keywords**

Declaring variables

- Variables that contain data (numbers, characters etc.) are the fundamental building blocks of any language
- Variables in Python are dynamically typed (see later)
- You can name your variables pretty much anything you want but:
  - Be **consistent** in your naming convention (see PEP 8)
  - Python is **case sensitive**; `genename` and `geneName` are different variables
  - First character cannot contain a digit: `1genename` raises an error
  - You cannot use Python's **reserved words/keywords**
  - If you're an R user, **DO NOT** use '.' in your variable names i.e `gene.name` is not a valid variable name

Core data types

- **Integers:** `int`

```
1 NGenes = 1500 # number of genes measured
2 type(NGenes) # int
```

Core data types

- **Integers:** `int`

```
1 NGenes = 1500 # number of genes measured
2 type(NGenes) # int
```

- **Floating point:** `float`**, any real numbers**

```
1 Km = 0.015 # Michaelis constant for chymotrypsin
2 type(Km) # float
```

Core data types

- **Integers:** `int`

```
1 NGenes = 1500 # number of genes measured
2 type(NGenes) # int
```

- **Floating point:** `float`**, any real numbers**

```
1 Km = 0.015 # Michaelis constant for chymotrypsin
2 type(Km) # float
```

- **Complex numbers:** `complex`

```
1 cnumber = 10 + 2j # 10=real part; 2=complex part
2 type(cnumber) # complex
```

Core data types (continued)

- **Boolean:** `bool` true or false values for logical statements

```
1 isTransFactor = True # is protein a transcription
      factor?
2 type(isTransFactor) # bool
```

Core data types (continued)

- **Boolean:** `bool` true or false values for logical statements

```
1 isTransFactor = True # is protein a transcription
      factor?
2 type(isTransFactor) # bool
```

- **Nothing:** `NoneType` when a variable is empty

```
1 moneyAmount = None
2 type(moneyAmount) #NoneType
```

Core data types (continued)

- **Boolean:** `bool` true or false values for logical statements

```
1 isTransFactor = True # is protein a transcription
      factor?
2 type(isTransFactor) # bool
```

- **Nothing:** `NoneType` when a variable is empty

```
1 moneyAmount = None
2 type(moneyAmount) #NoneType
```

- **Strings:** `str` for any text

```
1 motif = "AATCAGTT" # DNA sequence motif
2 type(motif) # str
```

Dynamic typing

- The type of a variable changes dependent on its value:

```
1 sequence = "AATCAGTT" # DNA sequence
2 type(sequence) # str
```

Dynamic typing

- The type of a variable changes dependent on its value:

```
1 sequence = "AATCAGTT" # DNA sequence
2 type(sequence) # str
3
4 sequence = 123456 # a numerical sequence
5 type(sequence) # int
```

Dynamic typing

- The type of a variable changes dependent on its value:

```
1 sequence = "AATCAGTT" # DNA sequence
2 type(sequence) # str
3
4 sequence = 123456 # a numerical sequence
5 type(sequence) # int
```

- This is also called duck typing *"If it walks like a duck and it quacks like a duck, then it must be a duck"*
  - i.e., we do not have to define from the start what data types we need (compare C or Java).
  - We find out along the way (e.g., by using `type`) what data types we are using

Container data types

- **Lists:** `list` for a collection of variables

```
1 # Interesting genes
2 geneNames = ["Irf1", "Ccl3", "Il12rb1", "Ifng", "Cxcl10
      "]
3 type(geneNames) # list
4
```

Container data types

- **Lists:** `list` for a collection of variables

```
1 # Interesting genes
2 geneNames = ["Irf1", "Ccl3", "Il12rb1", "Ifng", "Cxcl10
      "]
3 type(geneNames) # list
4
5 # Get elements of list
6 geneNames[0] # "Irf1"
7 geneNames[-1] # "Cxc110"
```

Container data types

- **Lists:** `list` for a collection of variables

```
1 # Interesting genes
2 geneNames = ["Irf1", "Ccl3", "Il12rb1", "Ifng", "Cxcl10
      "]
3 type(geneNames) # list
4
5 # Get elements of list
6 geneNames[0] # "Irf1"
7 geneNames[-1] # "Cxc110"
```

- **Tuples:** `tuple` for an *immutable* collection of values

```
1 # Interesting genes
2 geneNames = ("Irf1", "Ccl3", "Il12rb1", "Ifng", "Cxcl10
      ")
3 type(geneNames) # tuple
```

Initializing a single element tuple

- For some (daft) reason one might need a single element tuple. One might try:

```
1 singleElementT = ("Irf1")
2 type(singleElementT) # str, not a tuple!
```

Initializing a single element tuple

- For some (daft) reason one might need a single element tuple. One might try:

```
1 singleElementT = ("Irf1")
2 type(singleElementT) # str, not a tuple!
```

- Parentheses around a single element are having no effect (they are used for grouping). To clarify it is a tuple, one needs a trailing comma:

```
1 singleElementT = ("Irf1",)
2 type(singleElementT) # tuple
```

Container data types (continued)

- **Dictionary:** `dict` for a collection of values and unique labels

```
1 # A phone book
2 phoneBook = {"Bram": "01326 - 259022", "Annette": "
      01326 - 371842", "Angus": "01326 - 255794"}
3 type(phoneBook) # dict
```

Container data types (continued)

- **Sets:** `set`, `frozenset` for a collection of unique values

```
1 # Sets are mutable collections of unique values
2 languages = set(["Python", "R", "MATLAB", "C"])
3 type(languages) # set
```

Container data types (continued)

- **Sets:** `set`, `frozenset` for a collection of unique values

```
# Sets are mutable collections of unique values
languages = set(["Python", "R", "MATLAB", "C"])
type(languages) # set

# Frozensets are immutable collections of unique values
languages = frozenset(["Python", "R", "MATLAB", "C"])
type(languages) # frozenset
```

Container data types (continued)

- **Sets:** `set, frozenset` for a collection of unique values

```
# Sets are mutable collections of unique values
languages = set(["Python", "R", "MATLAB", "C"])
type(languages) # set

# Frozensets are immutable collections of unique values
languages = frozenset(["Python", "R", "MATLAB", "C"])
type(languages) # frozenset
```

- **Range:** `range` for sequences of integers

```
# Create immutable sequence of numbers from 0 to 4
x = range(5)
type(x) # range
```

Mutable vs immutable objects

- **Mutable** objects (`list, dict, set`) can be changed once assigned

```
1 # Lists are mutable
2 geneList = ["Irf1", "Ccl3", "Il12rb1"]
3 geneList[0]="Irf2" # change first gene to Irf2
```

Mutable vs immutable objects

- **Mutable** objects (`list, dict, set`) can be changed once assigned

```
1 # Lists are mutable
2 geneList = ["Irf1", "Ccl3", "Il12rb1"]
3 geneList[0]="Irf2" # change first gene to Irf2
```

- **Immutable** objects *cannot* be changed once assigned

```
1 geneTuple = ("Irf1", "Ccl3", "Il12rb1") # immutable
2 geneTuple[0]="Irf2"
3 TypeError: 'tuple' object does not support item
      assignment
```

Mutable vs immutable objects

- **Mutable** objects (`list, dict, set`) can be changed once assigned

```
1 # Lists are mutable
2 geneList = ["Irf1", "Ccl3", "Il12rb1"]
3 geneList[0]="Irf2" # change first gene to Irf2
```

- **Immutable** objects *cannot* be changed once assigned

```
1 geneTuple = ("Irf1", "Ccl3", "Il12rb1") # immutable
2 geneTuple[0]="Irf2"
3 TypeError: 'tuple' object does not support item
      assignment
```

- However you can replace an **immutable** object with a new one

```
1 geneTuple = ("Irf1", "Ccl3", "Il12rb1")
2 # Replace with a new object
3 geneTuple = ("Irf2", "Ccl3", "Il12rb1")
```

Why ever use immutable objects?

Immutable objects such as a `tuple` cannot be changed once assigned. Why use them?

Why ever use immutable objects?

Immutable objects such as a `tuple` cannot be changed once assigned. Why use them?

- It is faster: ideal if one only needs to iterate through a constant list of values

Why ever use immutable objects?

Immutable objects such as a `tuple` cannot be changed once assigned. Why use them?

- It is faster: ideal if one only needs to iterate through a constant list of values
- Assures that data is "write-protected" (i.e., elements cannot be changed after initialization)
  - Handy when one wants to ensure certain values are not changed during the course of a simulation

Making copies of objects

- In Python, copies are made by *reference*, not by *value*

Making copies of objects

- In Python, copies are made by *reference*, not by *value*
- Let us copy an `int`:

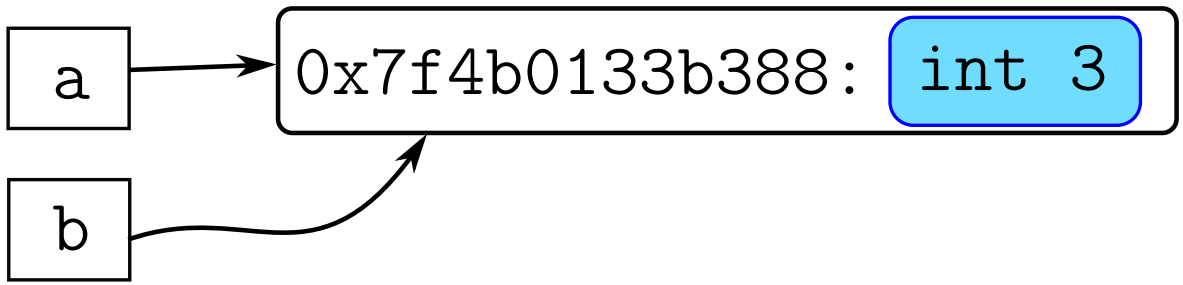
```
1 a = 3
2 b = a # copy of a
```

Making copies of objects

- In Python, copies are made by *reference*, not by *value*
- Let us copy an `int`:

```
1 a = 3
2 b = a # copy of a
```

- Both `a` and `b` then point to the same address in memory (say, `0x7f4b0133b388`), which holds `3`:



Making copies of objects

- So if we assign a new value to `a`, will `b` change too? No:

```
1 a = 5
2 print(a) # 5
3 print(b) # 3
```

Making copies of objects

- So if we assign a new value to `a`, will `b` change too? No:

```
1 a = 5
2 print(a) # 5
3 print(b) # 3
```

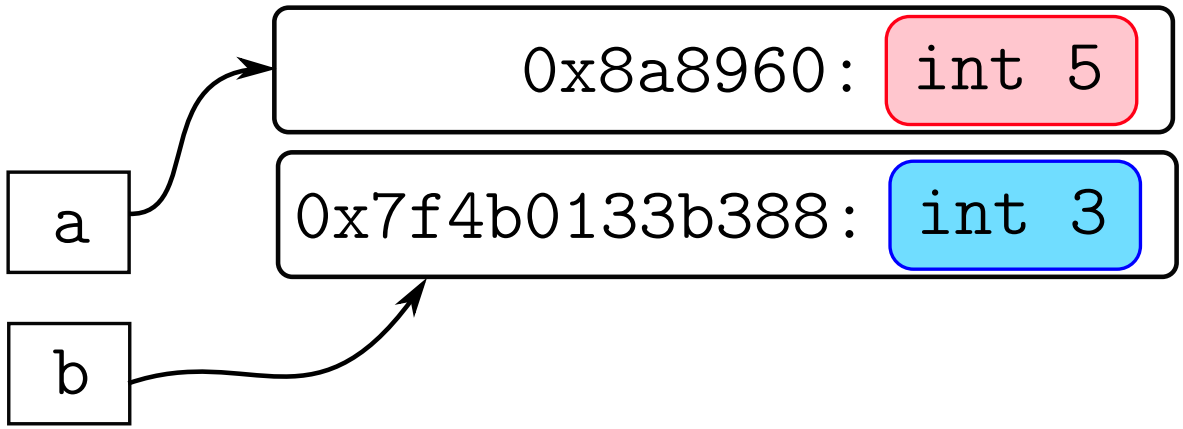
- What happens is that `a` is assigned a reference to a new position (0x8a8960) containing `5`
- While `b` keeps its reference to the old position (containing 3)

Making copies of objects

- So if we assign a new value to `a`, will `b` change too? No:

```
1 a = 5
2 print(a) # 5
3 print(b) # 3
```

- What happens is that `a` is assigned a reference to a new position (0x8a8960) containing `5`
- While `b` keeps its reference to the old position (containing 3)



Making copies of objects: lists

- What happens when making copies of objects which themselves contain objects (as happens in `list`, `dict`)?

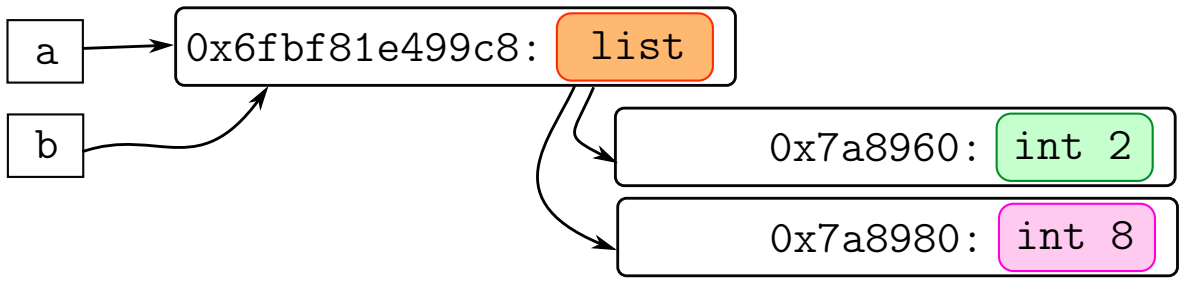
```
1 a = [2,8] # list with two ints
2 b = a # copy to the reference of that list
```

Making copies of objects: lists

- What happens when making copies of objects which themselves contain objects (as happens in `list`, `dict`)?

```
1 a = [2,8] # list with two ints
2 b = a # copy to the reference of that list
```

- Now our reference layout looks like this:



Making copies of objects: lists

- If we reassign a different variable to a list element, both `a` and `b` are affected!

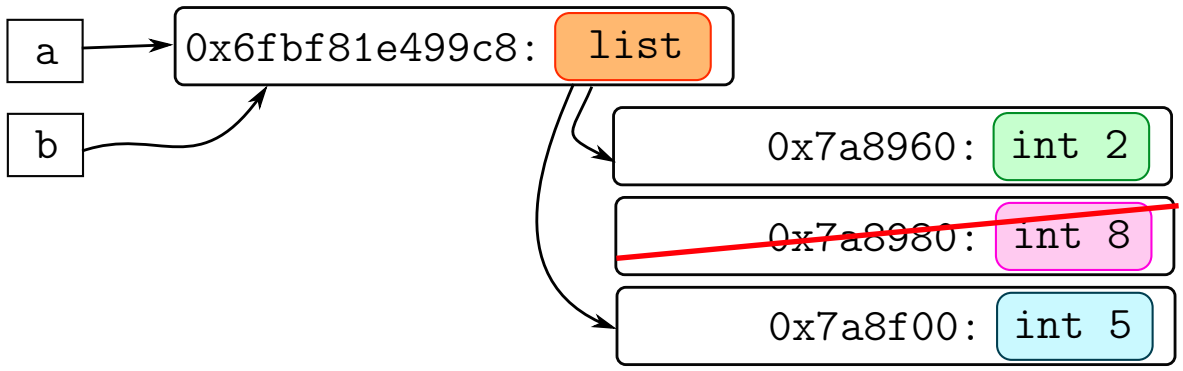
```
1 a[1] = 5
2 print(a) # [ 2, 5]
3 print(b) # [ 2, 5]
```

Making copies of objects: lists

- If we reassign a different variable to a list element, both `a` and `b` are affected!

```
1 a[1] = 5
2 print(a) # [ 2, 5]
3 print(b) # [ 2, 5]
```

- Both `b` and `a` still point to the same `list` object
- Within the list, however, references have now changed:



Making copies of objects: lists

- **Bottom line**: be careful when using copies of mutable objects like `list` or `dict`
- If you want to change elements only of the copied variable or not the original (but not both), make a *deep* copy:

```
1 a = [1,2]
2 b = a[:] # go through all elements of a and assign to b
```

Making copies of objects: lists

- **Bottom line**: be careful when using copies of mutable objects like `list` or `dict`
- If you want to change elements only of the copied variable or not the original (but not both), make a *deep* copy:

```
1 a = [1,2]
2 b = a[:] # go through all elements of a and assign to b
3
4 a[0] = 5
5 print(a) # [5,2]
6 print(b) # [1,2]: deep copy unaffected
```

Making copies of objects: lists

- **Bottom line**: be careful when using copies of mutable objects like `list` or `dict`
- If you want to change elements only of the copied variable or not the original (but not both), make a *deep* copy:

```
1  a = [1,2]
2  b = a[:] # go through all elements of a and assign to b
3
4  a[0] = 5
5  print(a) # [5,2]
6  print(b) # [1,2]: deep copy unaffected
7
8  # alternatively, use the copy module
9  import copy
10
11 b = copy.deepcopy(a)
```

Methods of objects

- A Python variable is called an **object**
- Every object has **methods** (functions) associated with it
- These methods are called using the dot notation (‘ . ’)

```
1 # DNA sequence motif
2 motif = "AATCAGTT"
3
4 # Use the count method to count occurrence of nucleotide "T"
5 motif.count("T") # 3
6
7 # Use the lower method to convert to lower case
8 motif.lower() # "aatcagtt"
```

Methods of objects

- A complete list of an object's methods can be found using the `__dir__()` function:

```
1 motif.__dir__()
2 # long list of methods associated to
```