python-variables-collections

October 16, 2019

1 Python For Data Science

Felix Biessmann

Lecture 2: Variables, Types, Operators and Data Structures

2 Python Variables

- are assigned with `=`
- are *dynamically typed* (have no static type)
- are pointers

2.1 Python Variables are assigned with =

[1]:
```
# assign 4 to the variable x
x = 4
print(x)
```

```
4
```

2.2 Python Variables are 'dynamically typed'

[2]:
```
x = 1         # x is an integer
print(x)
x = 'hello'   # now x is a string
print(x)
x = [1, 2, 3] # now x is a list
print(x)
```

```
1
hello
[1, 2, 3]
```

2.3 Dynamic Typing: Caveats

- Type only known at runtime
- Can result in lots of duck typing - or errors

"If it walks like a duck and it quacks like a duck, then it must be a duck"

2.3.1 Duck Typing

- 'Normal' (that is *static*) typing: variable is declared to be of certain type

```
int c = 0;
```

- Python does not have variables with static types (but see mypy)
- The **Duck Test** determines whether a variable can be used for a purpose to determine its type

[3]:
```python
# a function that for multiplying numbers by two:
def multiply_by_two(x):
    return x * 2

x = 2 # x is an integer
print(multiply_by_two(x))
```

```
4
```

[4]:
```python
x = "2" # x is a string
print(multiply_by_two(x))
```

```
22
```

2.4 How to check types by introspection

[5]:
```python
x = 1          # x is an integer
type(x)
```

[5]: `int`

[6]:
```python
x = "1"          # x is a string
type(x)
```

[6]: `str`

[7]:
```python
x = [1]          # x is a list
type(x)
```

[7]: `list`

[8]:
```python
x = [1]          # x is a list
issubclass(type(x), list)
```

[8]: True

3 Python Scalar Types

Type	Example	Description
`int`	`x = 1`	integers (i.e., whole numbers)
`float`	`x = 1.0`	floating-point numbers (i.e., real numbers)
`complex`	`x = 1 + 2j`	Complex numbers (i.e., numbers with real and imaginary part)
`bool`	`x = True`	Boolean: True/False values
`str`	`x = 'abc'`	String: characters or text
`NoneType`	`x = None`	Special object indicating nulls

3.1 Integers

[9]:
```python
x = 1
type(x)
```

[9]: int

[10]:
```python
# python ints are automatically casted to floats
x / 2
```

[10]: 0.5

3.2 Floats

[11]:
```python
x = 1.
type(x)
```

[11]: float

[12]:
```python
# explicit cast
x = float(1)
type(x)
```

[12]: float

[13]:
```
# equality checks between floats and ints actually work
x == 1
```

[13]:
```
True
```

3.2.1 Exponential notation

`e` or `E` can be read "…times ten to the…", so that `1.4e6` is interpreted as 1.4×10^6

[14]:
```
x = 1400000.00
y = 1.4e6
print(x == y)
```

```
True
```

3.3 None Type

[15]:
```
return_value = print('abc')
type(return_value)
```

```
abc
```

[15]:
```
NoneType
```

3.4 Boolean Type

- `True` and `False`
- Case sensitive!

[16]:
```
result = (4 < 5)
result
```

[16]:
```
True
```

[17]:
```
type(result)
```

[17]:
```
bool
```

3.4.1 Many types are implicitly cast to booleans:

3.4.2 Numbers

[18]:
```
bool(2014)
```

[18]: True

[19]:
```
bool(0)
```

[19]: False

3.4.3 None Types (or any other Type)

[20]:
```
bool(None)
```

[20]: False

3.4.4 Strings

[21]:
```
bool("")
```

[21]: False

[22]:
```
bool("abc")
```

[22]: True

3.4.5 Lists

[23]:
```
bool([1, 2, 3])
```

[23]: True

[24]:
```
bool([])
```

[24]: False

3.5 Python variables are pointers

[25]:
```
x = [1, 2, 3]
y = x
print(x)
print(y)
```

```
[1, 2, 3]
[1, 2, 3]
```

[26]:
```
# let's change the original variable x
x.append(4)
# now lets inspect y
print(y)
```

```
[1, 2, 3, 4]
```

[27]:
```
# however:
x = "Something entirely different"
print(y)
```

```
[1, 2, 3, 4]
```

3.6 Python variables are objects

- Objects (in all object oriented languages) have:
- Attributes / Fields
- Functions / Methods
- For simple types (`int`, `str`, …), many methods are accessible through **Operators**

4 Operators

- Arithmetic Operators
- Bitwise Operators
- Assignment Operators
- Comparison Operators
- Boolean Operators
- Membership Operators

4.1 Arithmetic Operations

Operator	Name	Description
`a + b`	Addition	Sum of `a` and `b`
`a - b`	Subtraction	Difference of `a` and `b`

Operator	Name	Description
`a * b`	Multiplication	Product of `a` and `b`
`a / b`	True division	Quotient of `a` and `b`
`a // b`	Floor division	Quotient of `a` and `b`, removing fractional parts
`a % b`	Modulus	Integer remainder after division of `a` by `b`
`a ** b`	Exponentiation	`a` raised to the power of `b`
`-a`	Negation	The negative of `a`
`+a`	Unary plus	`a` unchanged (rarely used)

[28]:
```python
a = 1
b = 1
a + b
```

[28]: 2

4.2 Bitwise Operations

Operator	Name	Description
`a & b`	Bitwise AND	Bits defined in both `a` and `b`
a | b	Bitwise OR	Bits defined in `a` or `b` or both
`a ^ b`	Bitwise XOR	Bits defined in `a` or `b` but not both
`a << b`	Bit shift left	Shift bits of `a` left by `b` units
`a >> b`	Bit shift right	Shift bits of `a` right by `b` units
`~a`	Bitwise NOT	Bitwise negation of `a`

[29]:
```python
a = True
b = False
a & b
```

[29]: False

4.3 Assignment Operations

`a += b`	`a -= b`	`a *= b`
`a //= b`	`a %= b`	`a **= b`
a |= b	`a ^= b`	`a <<= b`

[30]:
```
a = 2
a += 2  # equivalent to a = a + 2
print(a)
```

```
4
```

4.4 Boolean Operations

Operator	Description
`a and b`	True if `a` and `b`
`a or b`	True if `a` or `b` is true
`not a`	True if `a` is false.

[31]:
```
True and False
```

[31]:
```
False
```

[32]:
```
[True, True] or [False, True]
```

[32]:
```
[True, True]
```

[33]:
```
[True, True] and [False, 2]
```

[33]:
```
[False, 2]
```

4.5 Comparison Operations

Operation	Description	Operation	Description
`a == b`	`a` equal to `b`	`a != b`	`a` not equal to `b`
`a < b`	`a` less than `b`	`a > b`	`a` greater than `b`
`a <= b`	`a` less than or equal to `b`	`a >= b`	`a` greater than or equal to `b`

[34]:
```
# 25 is odd
25 % 2 == 1
```

[34]:
```
True
```

[35]:
```
# check if a is between 15 and 30
a = 25
15 < a < 30
```

[35]:
```
True
```

[36]:
```
# comparisons on standard collections are not element wise
[1,3] == [2,2]
```

[36]: False

[37]:
```
# but
a = [1,2]
b = a
a == b
```

[37]: True

4.6 Identity and Membership Operators

Operator	Description
`a is b`	True if `a` and `b` are identical objects
`a is not b`	True if `a` and `b` are not identical objects
`a in b`	True if `a` is a member of `b`
`a not in b`	True if `a` is not a member of `b`

[38]:
```
1 in [1,2,3]
```

[38]: True

4.7 Strings

- Python is great for Strings
- String encoding is a good reason to not use Python 2
- We'll do a quick recap of regexps

4.7.1 Some Useful String Functions

[39]:
```
message = "The answer is "
answer = '42'
```

[40]:
```
# length of string
len(answer)
```

[40]: 2

[41]:
```
# Make upper-case. See also str.lower()
message.upper()
```

[41]: 'THE ANSWER IS '

[42]:
```python
# concatenation
message + answer
```

[42]: 'The answer is 42'

[43]:
```python
# multiplication
answer * 3
```

[43]: '424242'

[44]:
```python
# Accessing individual characters (zero-based indexing)
message[0]
```

[44]: 'T'

[45]:
```python
# multiline strings
multiline_string = """
Computers are useless.
They can only give you answers.
"""
```

[46]:
```python
# stripping off unnecessary blanks (including \n or \t)
line = '         this is the content          '
line.strip()
```

[46]: 'this is the content'

[47]:
```python
# finding substrings
line = 'the quick brown fox jumped over a lazy dog'
line.find('fox')
```

[47]: 16

[48]:
```python
line.find('bear')
```

[48]: -1

[49]:
```python
# simple replacements
line.replace('brown', 'red')
```

[49]: 'the quick red fox jumped over a lazy dog'

[50]:
```python
# splitting a sentence into words
line.split()
```

[50]: ['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'a', 'lazy', 'dog']

[51]:
```python
# joining them back together
'--'.join(line.split())
```

[51]:
```
'the--quick--brown--fox--jumped--over--a--lazy--dog'
```

4.7.2 String Formatting

[52]:
```python
x = 3.14159
"This is a bad approximation of pi: " + str(x)
```

[52]:
```
'This is a bad approximation of pi: 3.14159'
```

[53]:
```python
"This is a bad approximation of pi: {}".format(x)
```

[53]:
```
'This is a bad approximation of pi: 3.14159'
```

[54]:
```python
y = 3
"This is a bad approximation of pi: {} but better than {}".format(x, y)
```

[54]:
```
'This is a bad approximation of pi: 3.14159 but better than 3'
```

[55]:
```python
"This is a bad approximation of pi: {1} but better than {0}".format(y, x)
```

[55]:
```
'This is a bad approximation of pi: 3.14159 but better than 3'
```

[56]:
```python
"""This is a bad approximation of pi: {bad} but
   better than {worse}""".format(bad=x, worse=y)
```

[56]:
```
'This is a bad approximation of pi: 3.14159 but \n   better than 3'
```

[57]:
```python
"Both of these approximations of pi are bad: {0:.3f} and {1}".format(x,y)
```

[57]:
```
'Both of these approximations of pi are bad: 3.142 and 3'
```

4.7.3 Since Python 3.6: f-String interpolation

[58]:
```python
width = 5
precision = 3
f'Bad approximation, nice f-string interpolation formatting: {x:{width}.
 ↪{precision}}'
```

[58]:
```
'Bad approximation, nice f-string interpolation formatting:  3.14'
```

4.7.4 Regular Expressions Recap

[59]:
```
import re
line = 'the quick brown fox jumped over a lazy dog'
regex = re.compile('fox')
match = regex.search(line)
match.start()
```

[59]:
```
16
```

[60]:
```
regex.sub('BEAR', line)
```

[60]:
```
'the quick brown BEAR jumped over a lazy dog'
```

4.7.5 Some Special Regexp Characters

Character	Description	Character	Description
"\d"	Match any digit	"\D"	Match any non-digit
"\s"	Match any whitespace	"\S"	Match any non-whitespace
"\w"	Match any alphanumeric char	"\W"	Match any non-alphanumeric char

There are many more special regexp characters; for more details, see Python's regular expression syntax documentation.

[61]:
```
regex = re.compile(r'\w\s\w')
regex.findall('the fox is 9 years old')
```

[61]:
```
['e f', 'x i', 's 9', 's o']
```

4.7.6 Finding Any Character in Set

If the special symbols are not enough, you can define your own character sets

[62]:
```
regex = re.compile('[aeiou]')
regex.split('consequential')
```

[62]:
```
['c', 'ns', 'q', '', 'nt', '', 'l']
```

[63]:
```
regex = re.compile('[A-Z][0-9]')
regex.findall('1043879, G2, H6')
```

[63]:
```
['G2', 'H6']
```

[64]:
```
regex = re.compile('[A-Z][A-Z][0-9]')
regex.findall('1043879, G2, H6 AH9')
```

[64]:
```
['AH9']
```

4.7.7 Wildcards

Character	Description	Example
`?`	Match zero or one repetitions of preceding	`"ab?"` matches `"a"` or `"ab"`
`*`	Match zero or more repetitions of preceding	`"ab*"` matches `"a"`, `"ab"`, `"abb"`, `"abbb"`…
`+`	Match one or more repetitions of preceding	`"ab+"` matches `"ab"`, `"abb"`, `"abbb"`… but not `"a"`
`{n}`	Match `n` repetitions of preceding	`"ab{2}"` matches `"abb"`
`{m,n}`	Match between `m` and `n` repetitions of preceding	`"ab{2,3}"` matches `"abb"` or `"abbb"`

[65]:
```
regex = re.compile('[A-Z][A-Z][0-9]')
regex.findall('1043879, G2, H6 AH9')
```

[65]:
```
['AH9']
```

[66]:
```
regex = re.compile('[A-Z]{2}[0-9]')
regex.findall('1043879, G2, H6 AH9')
```

[66]:
```
['AH9']
```

[67]:
```
regex = re.compile('[A-Z]+[0-9]')
regex.findall('1043879, G2, H6 AH9')
```

[67]:
```
['G2', 'H6', 'AH9']
```

[68]:
```
regex = re.compile('[A-Z]*[0-9]')
regex.findall('1043879, G2, H6 AH9')
```

[68]:
```
['1', '0', '4', '3', '8', '7', '9', 'G2', 'H6', 'AH9']
```

4.7.8 Example: Matching E-Mail Adresses

[69]:
```
email = re.compile('\w+@\w+\.[a-z]{3}')
text = "To email me, try user1214@python.org or hans@google.com."
email.findall(text)
```

[69]: ['user1214@python.org', 'hans@google.com']

[70]:
```
email.findall('barack.obama@whitehouse.gov')
```

[70]: ['obama@whitehouse.gov']

[71]:
```
email2 = re.compile(r'[\w.]+@\w+\.[a-z]{3}')
email2.findall('barack.obama@whitehouse.gov')
```

[71]: ['barack.obama@whitehouse.gov']

4.7.9 Matching Groups

Often it can be helpful to extract groups of matched substrings

[73]:
```
email3 = re.compile(r'([\w.]+)@(\w+)\.([a-z]{3})')
email3.findall(text)
```

[73]: [('user1214', 'python', 'org'), ('hans', 'google', 'com')]

4.7.10 Matching Named Groups

For programmatic treatment of groups, naming them can be useful

[74]:
```
email4 = re.compile(r'(?P<user>[\w.]+)@(?P<domain>\w+)\.(?P<suffix>[a-z]{3})')
match = email4.match('guido@python.org')
match.groupdict()
```

[74]: {'user': 'guido', 'domain': 'python', 'suffix': 'org'}

5 Data Structures

5.1 Builtin Python Data Structures

Type Name	Example	Description
list	[1, 2, 3]	Ordered collection
tuple	(1, 2, 3)	Immutable ordered collection

Type Name	Example	Description
`dict`	`{'a':1, 'b':2, 'c':3}`	Unordered (key,value) mapping
`set`	`{1, 2, 3}`	Unordered collection of unique values

5.2 Lists

- Ordered, indexable
- zero-based indexing
- Mutable
- Defined by `[1, 2, 3]`

5.2.1 List Indexing - Accessing Single Elements

[75]:
```
L = [2, 3, 5, 7, 11]
L[0]
```

[75]: 2

[76]:
```
L[1]
```

[76]: 3

[77]:
```
L[-1]
```

[77]: 11

[78]:
```
L[-2]
```

[78]: 7

5.2.2 List Slicing - Accessing Multiple Elements

[79]:
```
L[0:3]
```

[79]: [2, 3, 5]

[80]:
```
L[:3]
```

[80]: [2, 3, 5]

[81]:
```
L[-3:]
```

[81]: [5, 7, 11]

[82]:
```
L[-3:-1]
```

[82]:
```
[5, 7]
```

[83]:
```
L[::2]   # equivalent to L[0:len(L):2]
```

[83]:
```
[2, 5, 11]
```

[84]:
```
L[::-1] # reverses a list
```

[84]:
```
[11, 7, 5, 3, 2]
```

5.2.3 List Indexing and Slicing for Accessing and Assigning Elements

[85]:
```
L[0] = 100
L
```

[85]:
```
[100, 3, 5, 7, 11]
```

[86]:
```
L[1:3] = [55, 56]
L
```

[86]:
```
[100, 55, 56, 7, 11]
```

5.3 Lists

Operation	Example	Class
Access	`l[i]`	O(1)
Change Element	`l[i] = 0`	O(1)
Slice	`l[a:b]`	O(b-a)
Extend	`l.extend(...)`	O(len(…))
check ==, !=	`l1 == l2`	O(N)
Insert	`l[a:b] = ...`	O(N)
Delete	`del l[i]`	O(N)
Membership	`x in/not in l`	O(N)
Extreme value	`min(l)/max(l)`	O(N)
Multiply	`k*l`	O(k N)

Source

5.4 Tuples

- Similar to lists

- Immutable
- Defined by (1, 2, 3) or 1, 2, 3

[87]:
```python
t = (1, 2, 3)
t
```

[87]: (1, 2, 3)

[88]:
```python
t = 1, 2, 3
t
```

[88]: (1, 2, 3)

[89]:
```python
len(t)
```

[89]: 3

5.4.1 Elements cannot be changed

```python
t[0] = 5
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-86-6dd06f73cec4> in <module>()
----> 1 t[0] = 5

TypeError: 'tuple' object does not support item assignment
```

5.4.2 Return types of functions are often tuples

[90]:
```python
x = 0.125
x.as_integer_ratio()
```

[90]: (1, 8)

[91]:
```python
numerator, denominator = x.as_integer_ratio()
numerator / denominator
```

[91]: 0.125

5.5 Sets

- Unordered collections of unique items
- Support set operations
- Defined by {1, 2, 3}

[92]:
```
primes = {2, 3, 5, 7}
odds = {1, 3, 5, 7, 9}
```

5.5.1 Union

items appearing in either set

[93]:
```
primes = {2, 3, 5, 7}
odds = {1, 3, 5, 7, 9}
primes | odds       # with an operator
primes.union(odds) # equivalently with a method
```

[93]: {1, 2, 3, 5, 7, 9}

5.5.2 Intersection

items appearing in both sets

[94]:
```
primes = {2, 3, 5, 7}
odds = {1, 3, 5, 7, 9}
primes & odds              # with an operator
primes.intersection(odds) # equivalently with a method
```

[94]: {3, 5, 7}

5.5.3 Difference

items appearing in one but not other set

[95]:
```
primes = {2, 3, 5, 7}
odds = {1, 3, 5, 7, 9}
primes - odds            # with an operator
primes.difference(odds) # equivalently with a method
```

[95]: {2}

5.5.4 Symmetric difference

items appearing in only one set

[96]:
```
primes = {2, 3, 5, 7}
odds = {1, 3, 5, 7, 9}
primes ^ odds                        # with an operator
primes.symmetric_difference(odds) # equivalently with a method
```

[96]: {1, 2, 9}

5.6 Dictionaries

- Hash table
- Extremely flexible and versatile
- Fast access
- Unordered
- Defined by `key:value` pairs within curly braces: `{'a':1, 'b':2, 'c':3}`

[97]:
```
numbers = {'one':1, 'two':2, 'three':3}
```

[98]:
```
# Access a value via the key
numbers['two']
```

[98]: 2

[99]:
```
# Set a new key:value pair
numbers['ninety'] = 90
numbers
```

[99]: {'one': 1, 'two': 2, 'three': 3, 'ninety': 90}

5.7 Dictionary

Operation	Example	Class
Access	`d[k]`	O(1)
Change Element	`d[k] = 0`	O(1)
Delete	`del d[k]`	O(1)

Source