Strings, Collections, and Iteration

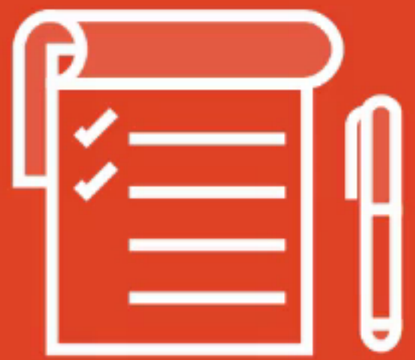
Austin Bingham
COFOUNDER - SIXTY NORTH

@austin_bingham

Robert Smallshire
COFOUNDER - SIXTY NORTH

@robsmallshire

Overview



`str`, `bytes`, `list`, and `dict`

`for`-loop

Put it all together

Collections

Collections

`str`

`bytes`

`list`

`dict`

`for-loops`

1) Strings in python have the data type "str"

2) Strings are **sequences of Unicode code points**, and for the most part, you can think of code points as being like characters, Although they are not strictly equivalent.

3) The **sequence of characters and a python string is immutable**, meaning that once you've constructed a string, you can't modify its contents.

4) Literal strings and python order limited by quotes. **You could use single quotes or double quotes**. You must, however, be consistent. For example, you can't use single quotes on one side and double on the other like this.

5) Supporting both quoting styles allows you to easily incorporate the other quote character into the literal string without resorting to ugly escape character gymnastics.
I noticed that the REPL exploits the same, quoting flexibility when echoing the strings back to us.

str

Data type for strings in Python

Sequence of Unicode code points

Immutable

String Literals

```
>>> 'This is a string'
'This is a string'
>>> "This is also a string"
'This is also a string'
>>> "inconsistent'
  File "<stdin>", line 1
    "inconsistent'
                  ^
SyntaxError: EOL while scanning string literal
>>> "It's a good thing."
"It's a good thing."
>>> '"Yes!", he said, "I agree!"'
'"Yes!", he said, "I agree!"'
>>>
```

Moment of Zen

Practicality beats purity

Beautiful text strings
Rendered in literal form
Simple elegance

String Literals

```
>>> "first" "second"
'firstsecond'
>>>
```

Adjacent literal strings are concatenated by the python compiler into a single string

Strings with Newlines

Multiline strings

Spread the literal across multiple lines

Escape sequences

Embed escape sequences in a single-line literal

If you want a literal string containing new lines, you have two options.
1) Use multi line strings or
2) Use escape sequences.

Multiline Strings

```
>>> """This is
... a multiline
... string"""
'This is\na multiline\nstring'
>>> '''So
... is
... this.'''
'So\nis\nthis.'
>>> m = 'This string\nspans multiple\nlines'
>>> m
'This string\nspans multiple\nlines'
>>> print(m)
This string
spans multiple
lines
>>>
```

Multi line strings are delimited by three quote characters rather than one. Here's an example using three double quotes. Notice how, when the string is echoed back to us, the new lines are represented by the \n escape sequence. We can also use three single quotes.

As an alternative to using multiline quoting we can just embedd ,the control characters ourselves.

To get a better sense of what we're representing, We can use print to see the string.

Newlines and Operating Systems

Windows Carriage-return, line-feed

\r\n

Linux and macOS Carriage-return

\r

Universal Newlines

Python translates \n to the appropriate newline sequence for your platform

PEP 278: python.org/dev/peps/pep-0278/

If you're working on Windows, you might be thinking that new lines should be represented by the carriage return and New line couplet \r\t. There's no need to do that with Python, since Python3 has a feature called U**niversal New Line Support,** which translates from the simple backslash end to the native Newline sequence for your platform on input and output.

Escape Sequences

```
>>> "This is a \" in a string"
'This is a " in a string'
>>> 'This is a \' in a string'
"This is a ' in a string"
>>> 'This is a \" and a \' in a string'
'This is a " and a \' in a string'
>>> k = 'A \\ in a string'
>>> k
'A \\ in a string'
>>> print(k)
A \ in a string
>>>
```

Because backslash has special meaning to place a backslash in String , we escape the backslash with itself. To reassure ourselves that there really is only one backslash in that string, we can print it.

All Escape Sequences

Sequence	Meaning
\newline	Backslash and newline ignored
\\	Backslash (\)
\'	Single quotes (')
\"	Double quote (")
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)
\r	ASCII Carriage Return (CR)
\t	ASCII Horizontal Tab (TAB)
\v	ASCII Vertical Tab (VT)
\ooo	Character with octal value *ooo*
\xhh	Character with hex value *hh*
Only recognized in string literals	

Strings and python are what are called **sequence types**, which means they support certain common operations for querying sequences.

For example, we can access individual characters using square brackets with an integer zero based index.

Note that, in contrast to many programming languages, **there is no separate character type distinct from the string type.** The indexing operation We just use returns, a full blown string that contains a single character element

We can use the string constructor (str) to create string representations of other types, such as integers or floats.

String Features

```
>>> path = r'C:\Users\Merlin\Documents\Spells'
>>> path
'C:\\Users\\Merlin\\Documents\\Spells'
>>> print(path)
C:\Users\Merlin\Documents\Spells
>>> str(496)
'496'
>>> str(6.02e23)
'6.02e+23'
>>> s = 'parrot'
>>> s[4]
'o'
>>> type(s[4])
<class 'str'>
>>>
```

Raw Strings: Sometimes, particularly when dealing with strings such as windows file system paths or regular expression patterns, which use backslash extensively, the requirement to double up on back slashes could be ugly and error prone .

Python comes to the rescue with its **raw strings.**

Raw strings don't support any escape sequences and are very much what you see is what you get .

To create a raw string prefix the opening quote with a lower case r .

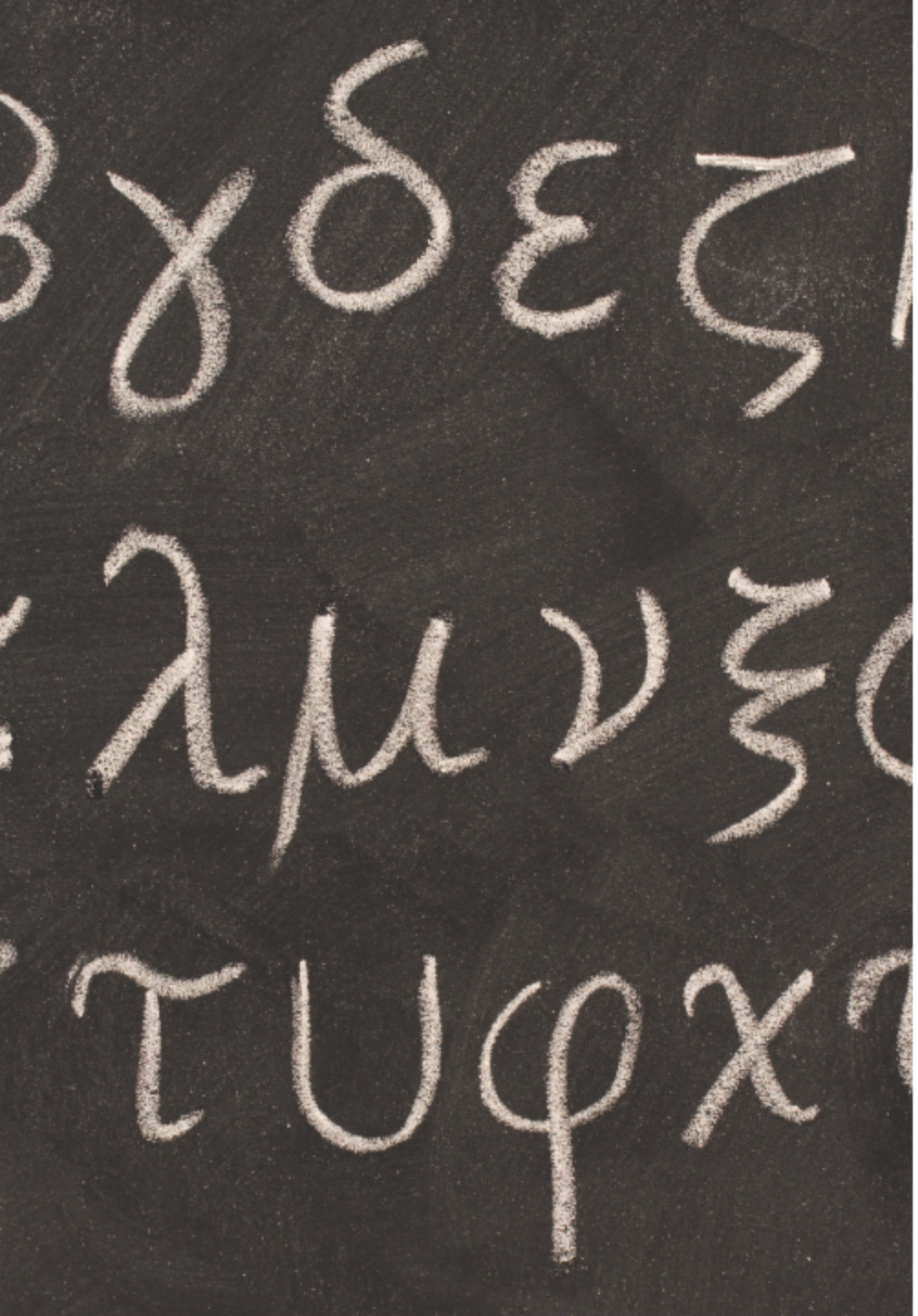
String Features

```
 |
 |  __sizeof__(self, /)
 |      Return the size of the string in memory, in bytes.
 |
 |  __str__(self, /)
 |      Return str(self).
 |
 |  capitalize(self, /)
 |      Return a capitalized version of the string.
 |
 |      More specifically, make the first character have upper case and the rest
lower
 |      case.
 |
 |  casefold(self, /)
 |      Return a version of the string suitable for caseless comparisons.
 |
```

```
>>> c = "oslo"
>>> c.capitalize()
'Oslo'
>>> c
'oslo'
>>>
```

In python, we use the dot after the object name and before the method Name. Methods are functions, so we must use the parentheses to indicate that the method should be called.

Remember that strings are immutable, so the capitalize method didn't modify the string in place. Rather, it returned a new string.



`str` is Unicode

Python 3 source encoding is UTF-8

Unicode in Strings

```
>>> "Vi er så glad for å høre og lære om Python!"
'Vi er så glad for å høre og lære om Python!'
>>> "Vi er s\u00e5 glad for \u00e5 h\xf8re og l\u00e6re om Python!"
'Vi er så glad for å høre og lære om Python!'
>>> '\xe5'
'å'
>>> '\345'
'å'
>>>
```

Because strings are fully Unicode capable, we can use them with international characters easily, even in literals. Because the default source code encoding for python 3 is utf-8. For example, if you have access to norvegian characters, you can simply enter as shown above.
Alternatively, you can use the Hexadecimal representations of Unicode code points as an escape sequence prefixed by \u.
Similarly, you can use the \x escape sequence, followed by a two character Hexadecimal string or an escaped Octal string to include Unicode characters in a string literal.

There are no such Unicode capabilities in the otherwise similar bytes type

Bytes are very similar to strings, except that rather than being sequences of unicode code points there sequences of bytes.

They're used for raw binary data and fixed with a single byte character encodings such as ASCII .

Their literal form is prefixed by b.

bytes

Data type for sequences of bytes

Raw binary data

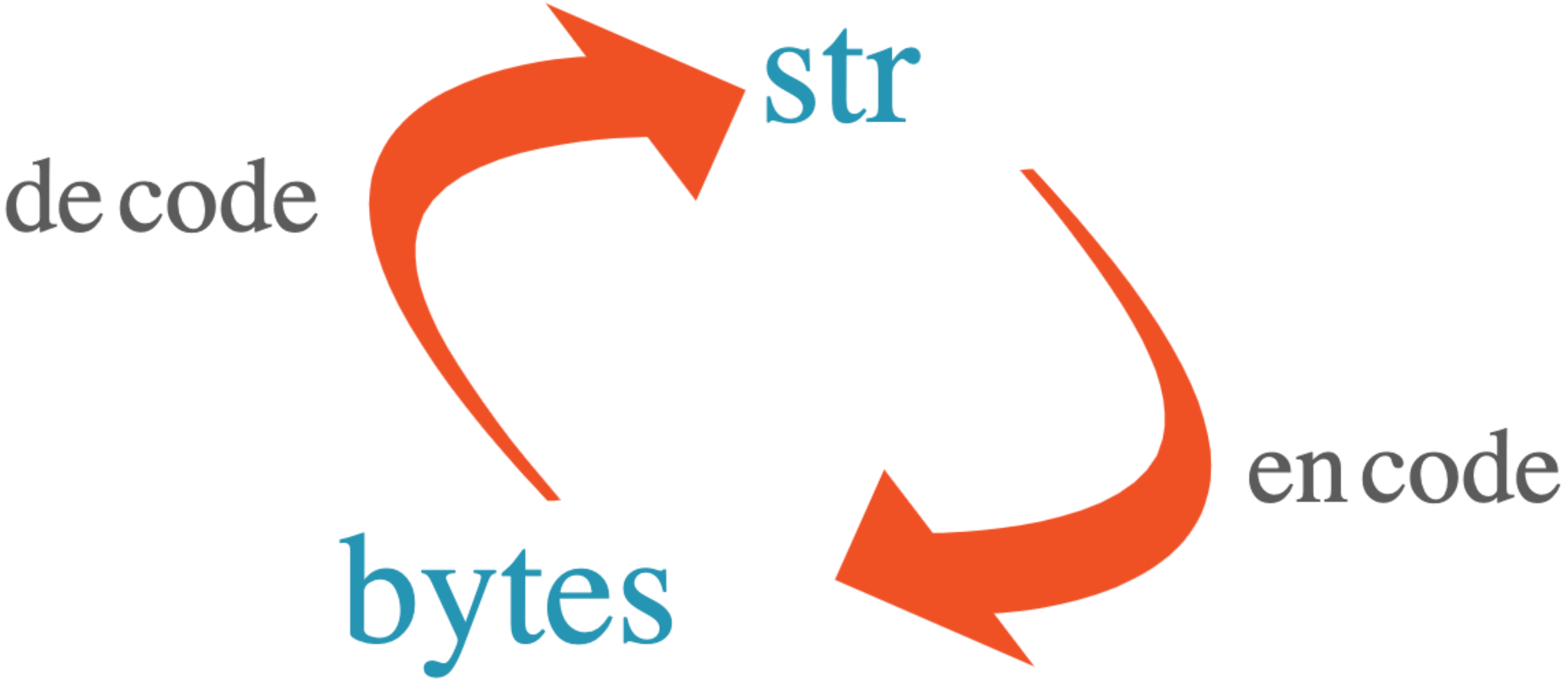
Fixed-width single-byte encodings

Bytes

They support most of the same operations as string, such as indexing, which returns the integer value of the specified byte and splitting, which you'll see returns a list of bytes objects

```
>>> b'data'
b'data'
>>> b"data"
b'data'
>>> d = b'some bytes'
>>> d[0]
115
>>> d.split()
[b'some', b'bytes']
>>>
```

Converting Betweeen Strings and Bytes



docs.python.org/3/library/codecs.html#standard-encodings

Encode/Decode

To convert between bytes and strings we must know the encoding of the byte sequence used to represent the strings unicode code points as bytes. Python supports a wide variety of encodings, a full list of which can be found at python.org.

```
>>> norsk = "Jeg begynte å fortære en sandwich mens jeg kjørte taxi på vei til quiz"
>>> data = norsk.encode('utf8')
>>> data
b'Jeg begynte \xc3\xa5 fort\xc3\xa6re en sandwich mens jeg kj\xc3\xb8rte taxi p\xc3\
xa5 vei til quiz'
>>> norwegian = data.decode('utf8')
>>> norwegian == norsk
True
>>> norwegian
'Jeg begynte å fortære en sandwich mens jeg kjørte taxi på vei til quiz'
>>>
```

1) Python lists, such as those returned by the string split method are sequences of objects.

2) Unlike strings. Lists are mutable in so far as the elements within them can be replaced or removed and new elements can be inserted or appended.

3) Lists are a workhorse of python data structures.

4) Literal lists are delimited by square brackets and the items within the list separated by commas.
5) We can retrieve elements by using square brackets with zero based index, and we can replace elements by assigning to a specific element.

6) Lists could be heterogeneous ,with respect to the types of the objects.

7) To create empty list use, empty square brackets.

list

Sequences of objects

Mutable

A workhorse in Python

Lists

```
>>> [1, 9, 8]
[1, 9, 8]
>>> a = ["apple", "orange", "pear"]
>>> a[1]
'orange'
>>> a[1] = 7
>>> a
['apple', 7, 'pear']
>>> b = []
>>> b.append(1.618)
>>> b
[1.618]
>>> b.append(1.414)
>>> b
[1.618, 1.414]
>>> list("characters")
['c', 'h', 'a', 'r', 'a', 'c', 't', 'e', 'r', 's']
>>> c = ['bear',
...      'giraffe',
...      'elephant',
...      'caterpillar',]
>>> c
['bear', 'giraffe', 'elephant', 'caterpillar']
>>>
```

If at the end of the line brackets, braces or parentheses are unclosed, you can continue on the next line.

This could be very useful for long, literal collections or simply to improve readability.

A list constructor, could be used to create lists from other collections such a strings.

We're allowed to use an additional comma after the last element. This is an important maintainability feature.

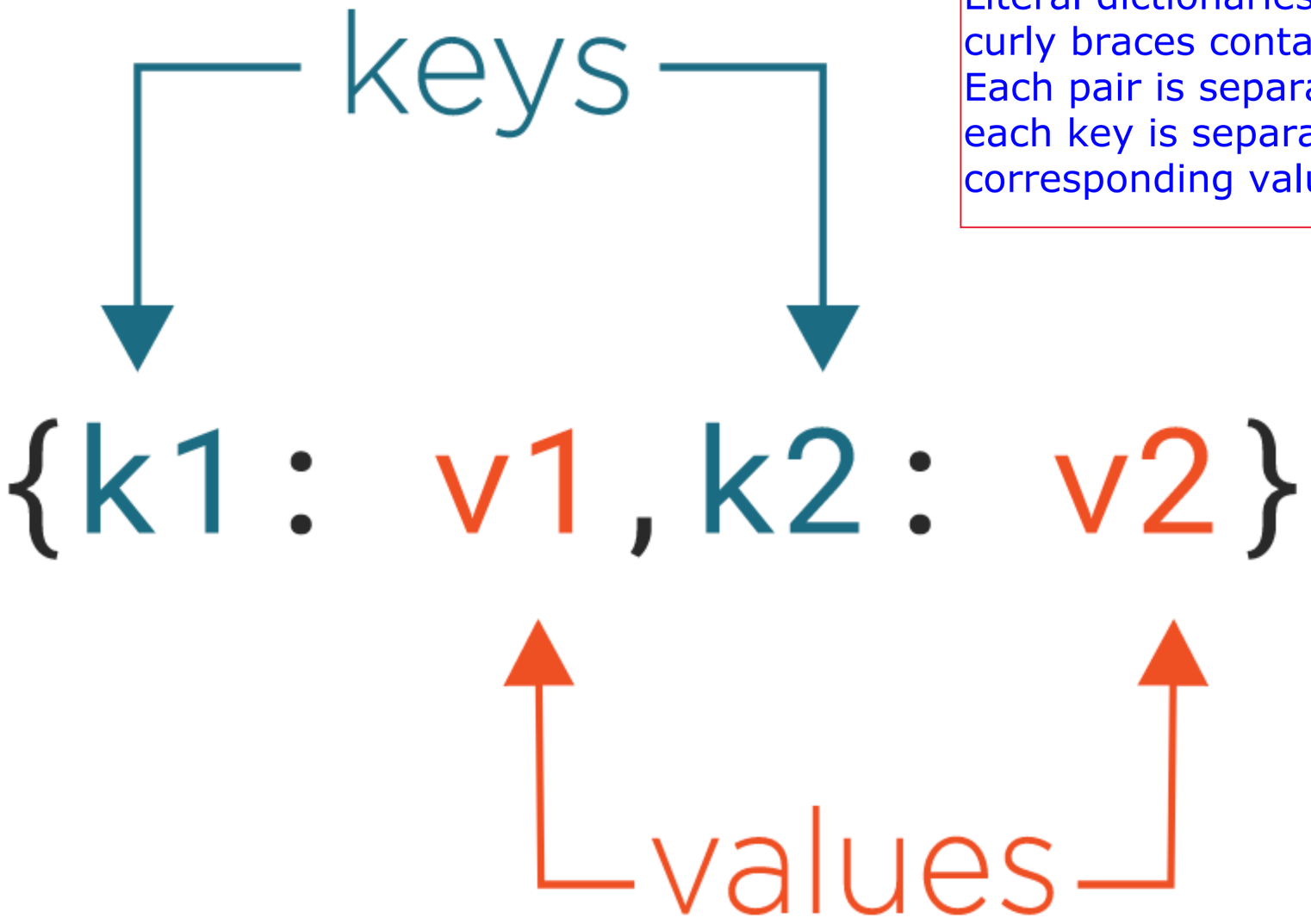
dict

Fundamental data structure in Python

Map keys to values

Also known as maps or associative arrays

Dict Literals



A dictionary maps, keys to values and in other languages, is known as a map or an associative array.

Literal dictionaries are created using curly braces containing key value pairs. Each pair is separated by a comma, and each key is separated from the corresponding value by a colon.

Dict

We could retrieve items by key, using the square brackets operator and update the values associated with the key by assigning through the square brackets.

If we assigned to a key that has not yet been added, a new entry is created.

```
>>> d = {'alice': '878-8728-922', 'bob': '256-5262-124', 'eve': '198-2321-787'}
>>> d['alice']
'878-8728-922'
>>> d['alice'] = '966-4532-6272'
>>> d
{'alice': '966-4532-6272', 'bob': '256-5262-124', 'eve': '198-2321-787'}
>>> d['charles'] = '334-5551-913'
>>> d
{'alice': '966-4532-6272', 'bob': '256-5262-124', 'eve': '198-2321-787', 'charle
s': '334-5551-913'}
>>> e = {}
>>>
```

Be aware that in python versions prior to 3.7, the entries in the dictionary can't be relied upon to be stored in any particular order.

As of Python 3.7, however, entries are required to be kept in insertion order.

Similar to lists empty dictionaries can be created using empty curly braces.

for-loop

Visit each item in an iterable sequence

The for loops in python correspond to what are called foreach loops in many other programming languages.

They request items one by one from a collection or more strictly from iterable Series and assign them in turn toe a variable that we specify.

For-loops

```
for item in iterable:
    ...body...
```

For-loop

```
>>> cities = ["London", "New York", "Paris", "Oslo", "Helsinki"]
>>> for city in cities:
...     print(city)
...
London
New York
Paris
Oslo
Helsinki
>>> colors = {'crimson': 0xdc143c, 'coral': 0xff7f50, 'teal': 0x008080}
>>> for color in colors:
...     print(color, colors[color])
...
crimson 14423100
coral 16744272
teal 32896
>>>
```

If you iterate over dictionaries, you get the keys, which you can then use within the for loop body to retrieve values.

Note that we used the ability of the built-in print function to accept multiple arguments. We passed the key and the value for each color separately. See also how the color codes return to us are in decimal

Putting It All Together

```
or', b'good', b'or', b'for', b'evil', b'in', b'the', b'superlative', b'degree',
b'of', b'comparison', b'only']
>>>
>>>
>>>
.decode('utf8').split()
...
...
...
>>>
>>>
['It', 'was', 'the', 'best', 'of', 'times', 'it', 'was', 'the', 'worst', 'of',
times', 'it', 'was', 'the', 'age', 'of', 'wisdom', 'it', 'was', 'the', 'age', 'o
f', 'foolishness', 'it', 'was', 'the', 'epoch', 'of', 'belief', 'it', 'was', 'th
e', 'epoch', 'of', 'incredulity', 'it', 'was', 'the', 'season', 'of', 'Light', '
it', 'was', 'the', 'season', 'of', 'Darkness', 'it', 'was', 'the', 'spring', 'of
', 'hope', 'it', 'was', 'the', 'winter', 'of', 'despair', 'we', 'had', 'everythi
ng', 'before', 'us', 'we', 'had', 'nothing', 'before', 'us', 'we', 'were', 'all'
, 'going', 'direct', 'to', 'Heaven', 'we', 'were', 'all', 'going', 'direct', 'th
e', 'other', 'way', 'in', 'short', 'the', 'period', 'was', 'so', 'far', 'like',
'the', 'present', 'period', 'that', 'some', 'of', 'its', 'noisiest', 'authoritie
s', 'insisted', 'on', 'its', 'being', 'received', 'for', 'good', 'or', 'for', 'e
vil', 'in', 'the', 'superlative', 'degree', 'of', 'comparison', 'only']
>>>
```

Notice that each of the single quoted words is prefixed by a lower case letter B, meaning that we have a list of bytes objects.

This is because the http request transferred raw bytes to us over the network.

To get a list of strings,we should decode the bytes stream in each line into Unicode strings. We can do this by **inserting a call to the decode method of the bytes object** and then operating on the resulting Unicode string.

Check for Errors

Carefully check and re-enter code if there are errors.

An `HTTPError` indicates a network problem.

Recall Bytes

Bytes literals prefixed with lowercase 'b'

HTTP data is provided as bytes

Use `bytes.decode()` to get strings

From REPL to IDE

The REPL is good for short-lived work and experimentation.

Use an editor/IDE for larger or longer-lived projects.

Adjacent string literals are implicitly concatenated.

Python has support for universal new lines. So no matter what platform you're using, it's sufficient to use a single \n character, safe in the knowledge that it will be appropriately translated from and to the native new line during Io

Escape sequences provide an alternative means of incorporating new lines and other control characters into literal strings.

The backslash is used for escaping, can be a hindrance for Windows file system paths or regular expressions, so raw strings with r prefix can be used to suppress the escaping mechanism.

In python-3 literal strings can contain Unicode characters directly in the source.

Strings

- Single- and multi-line literals
- Concatenation of adjacent literals
- Universal newlines
- Escape sequences
- Raw strings
- Use str constructor to convert other types
- Access individual characters with square bracket indexing
- Rich API
- String literals can contain Unicode

The bytes type has many of the capabilities of strings, but it is a sequence of bytes rather than a sequence of Unicode Code points.

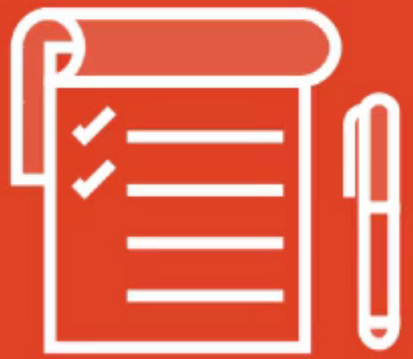
Bytes literals are prefixed by lowercase 'b'.

To convert between string and bytes instances ,we use encode method on strings and and the decode method of bytes. In both cases, passing the encoding, which we must know in advance.

Bytes

- Sequence of bytes rather than codepoints
- Literals prefixed with lowercase "b"
- Use `str.encode()` and `bytes.decode()` for conversion

Summary



Lists

- Mutable, heterogeneous sequences
- Literals delimited by square brackets
- Literal items separated by commas
- Access elements with square brackets
- Elements can be replaced by assigning to an index
- Grow lists with `append()`
- Use `list` constructor to create lists from other sequences

Summary

Dicts

- Associate keys with values
- Literals are delimited by curly braces
- Key-value pairs are separated by commas
- Keys are separated from values by colons

For-loops

- Bind each item from an iterable one at a time to a name
- Called for-each loops in other languages