01_Python_Introduction

October 29, 2019

1 Introduction to Python

What is python?

- Python is an interpreted language, invented in the 1990s by Guido van Rossum.

Which python version are we using?

- There are two major versions of Python: Python2 and Python3. Because support for Python2 will end in 2020, in this couse we'll be learning Python 3.7

How to learn python3?

- Learning a new programming language takes some amount of time, especially if you're new to programming. The best way to learn a language is via practice. You'll have a good amount of practice via homeworks and final projects, but if that's not enough https://www.codecademy.com/learn/learn-python-3 e.g. offers a repl-based way of learning python. Feel free to take advantage of their (free) course.

The material on this notebook is mostly based on

1.0.1 01.01 Comments

Comments in python are defined similarly to comments in bash via #

```
In [1]: # this is a comment in python

        """ Another option to define a comment is to use
        multiline string syntax e.g. """

        'or just a string object'

        # however, best is to use #
```

```
Out[1]: 'or just a string object'
```

1.0.2 01.02 Variables and basic operations

Everything in python is a variable:

1. numbers, strings
2. objects
3. lists, dictionaries, tuples, sets
4. functions

The type of a variable is defined during runtime. Python uses duck-typing, i.e. is strongly typed during runtime.

1.0.3 01.02.01 Number Expressions

The simplest expressions one can write are just using numbers. Note that the type changes depending what operations and whether integers, booleans or floats are involved

```
In [2]: 1 + 2

Out[2]: 3

In [3]: type(1+2)

Out[3]: int

In [4]: 1 + 2.

Out[4]: 3.0

In [5]: type(1+ 2.)

Out[5]: float
```

Booleans can be also added to numbers with the mapping `True=1`, `False=0`

```
In [6]: True

Out[6]: True

In [7]: False

Out[7]: False

In [8]: True + True

Out[8]: 2

In [9]: False + True

Out[9]: 1

In [10]: True + 2.7
```

```
Out[10]: 3.7
```

Python supports +, -, *, /. However, when using / the result will be always a float!

```
In [11]: 5 + 7

Out[11]: 12

In [12]: 5 - 7

Out[12]: -2

In [13]: 5 * 7

Out[13]: 35

In [14]: 5 / 7

Out[14]: 0.7142857142857143
```

In addition, python has operators for - ** power/exponentiation - // integer division - % modulo

```
In [15]: 5.0 // 2.0

Out[15]: 2.0

In [16]: 2 ** 0.5

Out[16]: 1.4142135623730951

In [17]: 7 ** 3

Out[17]: 343

In [18]: 3.7 % 2.0

Out[18]: 1.7000000000000002
```

More complex expressions can be made up using parentheses

```
In [19]: (1 + 2) * 7

Out[19]: 21

In [20]: # floating point numbers are 64bit (double)
         2. * 1.

Out[20]: 2.0

In [21]: # python integers support arbitrary precision, i.e. it's possible to write something
         2 ** 100

Out[21]: 1267650600228229401496703205376
```

1.1 01.02.02 String expressions

Strings can be declared using '' or "". To escape ', " respectively use \.

```
In [22]: 'This is a string'

Out[22]: 'This is a string'

In [23]: "can be also declared like this"

Out[23]: 'can be also declared like this'

In [24]: 'To escape \' use \\\''

Out[24]: "To escape ' use \\'"

In [25]: # One can also declare multiline string via ''' ... ''' or """ ... """

In [26]: """this
         is
         a
         multiline
         string"""

Out[26]: 'this\nis\na\nmultiline\nstring'

In [27]: '''this
         also'''

Out[27]: 'this\nalso'
```

To break up long strings or commands, can be used. NOTE: No comment or whitespace after !

```
In [28]: 'one cool way is' \
         'to break up strings via \\'

Out[28]: 'one cool way isto break up strings via \\'
```

This works, because string literals next to each other are automatically concatenated

```
In [29]: 'Hello ' 'world'

Out[29]: 'Hello world'
```

One can also perform certain operations on strings, like - + concatenation - * replication

```
In [30]: 'this ' + 'is ' + 'a ' + 'concatenated ' + 'string'

Out[30]: 'this is a concatenated string'

In [31]: 'ha' * 3

Out[31]: 'hahaha'
```

1.1.1 01.02.03 Variables

variables are simply declared using `name = value`. Their type is deducted at runtime from the type of the `value`

```
In [32]: # declaring a variable
         a = 10
         b = 4.5

In [33]: a * b

Out[33]: 45.0

In [34]: a / b

Out[34]: 2.2222222222222223

In [35]: a ** 2

Out[35]: 100

In [36]: a + b

Out[36]: 14.5

In [37]: a - b

Out[37]: 5.5

In [38]: a + 12

Out[38]: 22

In [39]: a % 3

Out[39]: 1

In [40]: a + 3.

Out[40]: 13.0
```

Python has also a special value called `None`, which can be used

```
In [41]: x=None

         x
```

1.2 01.03 Lists and tuples

Besides support for booleans, ints, floats and strings python has builtin support for lists and tuples
==> list: a mutable collection of objects (i.e. can remove, append)
==> tuple: an immutable collection of objects
To declare a list use `[...]`

```
In [42]: [1, 2, 3]

Out[42]: [1, 2, 3]
```

A list can store any object!

```
In [43]: ['hello', 'world', 12, 34, 3.141, 7.0/5]

Out[43]: ['hello', 'world', 12, 34, 3.141, 1.4]
```

A tuple can be declared using `(...)`

```
In [44]: (1, 2, 3)

Out[44]: (1, 2, 3)

In [45]: ('hello', 12)

Out[45]: ('hello', 12)
```

Basic indexing is done on both lists and tuples via `[...]` and zero-based

```
In [46]: t = (1, 2, 3)
         t[0]

Out[46]: 1

In [47]: L = [1, 2, 3]
         L[1]

Out[47]: 2
```

1.2.1 01.03.01 appending / removing from lists

To append another list to a list, you can use + or `.extend(...)`

```
In [48]: [1, 2, 3] + [4, 5]

Out[48]: [1, 2, 3, 4, 5]

In [49]: L=[1, 2, 3]
         L.extend([4, 5])
         L

Out[49]: [1, 2, 3, 4, 5]
```

Note that `extend` is a **void** function, i.e. has no return value and thus yields `None`

```
In [50]: type([1, 2, 3].extend([4, 5]))
```

```
Out[50]: NoneType
```

Similarly, to append a single element you can use `+ [el]` or `.append(el)`

```
In [51]: L = [1, 2, 3]
         L += [0]
         L
```

```
Out[51]: [1, 2, 3, 0]
```

```
In [52]: L.append(1)
         L
```

```
Out[52]: [1, 2, 3, 0, 1]
```

Tuples are immutable so there is no append/remove.
To delete a single element from a list, you can use `del L[i]`

```
In [53]: del L[2]
         L
```

```
Out[53]: [1, 2, 0, 1]
```

1.3 01.03.02 Checking whether element exists in list/tuple

Lists, tuples and strings are sequence types (with strings being immutable).
The `in` operator can be used to check whether an element exists within a list/tuple.
For strings, `in` checks whether a substring exists.

```
In [54]: 'substr' in 'does substring exist in this string?'
```

```
Out[54]: True
```

```
In [55]: 3 in [1, 2, 3, 4, 5]
```

```
Out[55]: True
```

```
In [56]: -1 in (1, 2, 3)
```

```
Out[56]: False
```

```
In [57]: # This doesn't work for lists/tuples.
         # it checks whether an element [2, 3] exists in a list
         [2, 3] in [1, 2, 3, 4, 5]
```

```
Out[57]: False
```

1.4 01.03.03 Slicing

Elements of sequence types can be either accessed via a single index or a range of indices, to obtain a slice.

Indices in python can be also negative to access elements from the rear. I.e. L[-1] yields the last element of the list.

```
In [58]: L = [1, 2, 3, 4]
         L[-1]

Out[58]: 4

In [59]: L[-4]

Out[59]: 1

In [60]: s = 'This is a string'
         s[-1]

Out[60]: 'g'

In [61]: t = (1, 2, 3)
         t[-2]

Out[61]: 2
```

The length of a sequence can be determined using `len(...)`, i.e.

```
In [62]: len(L)

Out[62]: 4

In [63]: len(t)

Out[63]: 3

In [64]: len(s)

Out[64]: 16
```

A slice can be specified using `:`. The syntax is thereby `start:end:step`. Each of `start, end, step` can be left out. I.e. if `step` is left out, it defaults to `1`. `start` to `0` and `end` to `len(...)`.

==> `start` index is inclusive, `end` index is exclusive!

```
In [65]: L = [1, 2, 3, 4, 5, 6, 7]
```

specifying no start, end or step will yield the full list

```
In [66]: L[::]

Out[66]: [1, 2, 3, 4, 5, 6, 7]
```

```
In [67]: L[2:]

Out[67]: [3, 4, 5, 6, 7]

In [68]: L[:-2]

Out[68]: [1, 2, 3, 4, 5]

In [69]: L[1:3:]

Out[69]: [2, 3]

In [70]: L[1:-1:2]

Out[70]: [2, 4, 6]
```

An especially useful case for reversing a sequence is to get the slice `[::-1]`

```
In [71]: L[::-1]

Out[71]: [7, 6, 5, 4, 3, 2, 1]

In [72]: s = 'This also works for strings'
         s[::-1]

Out[72]: 'sgnirts rof skrow osla sihT'
```

On lists, a slice can be also used to delete a sublist/substring.
Note: this works only for simple slices of the form `start:end`

```
In [73]: L

Out[73]: [1, 2, 3, 4, 5, 6, 7]

In [74]: L[2:5] = []
         L

Out[74]: [1, 2, 6, 7]
```

1.5 02.01 String formatting

Python offers a rich set of options to format strings, there are two standard ways - format new Python3 way of formatting strings via python's own mini language - C-like formatting using `%d`, `%f` similar to printf

```
In [75]: # print(...) converts an object to a string and prints it out with a newline to stdou

In [76]: print('Hello world')

Hello world
```

C style formatting: python offers the % operator on string for that:

```
In [77]: a = 42
         b = 17
         print('The result of %d + %d = %d' % (a, b, a + b))
```

```
The result of 42 + 17 = 59
```

You can use all the formatting options like in any C-language. For more on this, take a look at https://notgnoshi.github.io/printf/

```
In [78]: print('%04d or %.03f' % (10, 1 / 7))
```

```
0010 or 0.143
```

The new, pythonic way works via the format mini language (details under https://docs.python.org/3.7/library/string.html#format-examples or https://www.digitalocean.com/community/tutorials/how-to-use-string-formatters-in-python-3)

```
In [79]: 'the new pythonic way of formatting numbers like {:.3f} or integers {:04d} looks like
```

```
Out[79]: 'the new pythonic way of formatting numbers like 3.142 or integers -002 looks like thi
```

There are more ways of string formatting, but above are the two standard ways you should know.

1.6 3.01 Control flow

So far except for assigning variables (i.e. `x = 20`), we have dealt only with expressions. Statements make a language powerful! Python offers all the classical control flow (compound) statements:

Rule: 1 tab = 4 spaces

```
In [80]: x = 2

         if x < 0:
             print('x is negative') # note the whitespace to start a block of statements, shou
         else:
             print('x is non-negative')
```

```
x is non-negative
```

if statements also offer elif (elseif)

```
In [81]: animal = 'penguin'

         if animal == 'lion':
             print('big5!')
```

```
        elif animal == 'leopard':
            print('big5!')
        elif animal == 'rhinoceros':
            print('big5!')
        elif animal == 'elephant':
            print('big5!')
        elif animal == 'buffalo':
            print('big5!')
        else:
            print('just a small animal')
```

```
just a small animal
```

Instead of using if as a statement, it can be also used as expression

```
In [82]: number = 11
         number_info = 'even' if number % 2 == 0 else 'odd'

         number_info
```

```
Out[82]: 'odd'
```

Use intendation to nest statements

```
In [83]: coords = (0.5, 0.5)


         if 0.0 < coords[0] < 1.0:
             if 1.0 > coords[1] > 0.0:
                 print('inside unit square')
             else:
                 print('outside unit square')
         else:
             print('outside unit square')
```

```
inside unit square
```

Conditional expressions
Use `and` or `or` to combine multiple boolean expressions
Supported operators are `==`, `!=`, `<`, `>`, `<=`, `>=` (work also for strings!)

```
In [84]: x = 25

         if x < 100 and x >= 10:
             print('x can be represented using two digits in the decimal system')
```

```
x can be represented using two digits in the decimal system
```

Loops

Python has while and for loops.

```
In [85]: for i in [1, 2, 3, 4, 5]:
             print('2**{} = {}'.format(i, 2 ** i))

2**1 = 2
2**2 = 4
2**3 = 8
2**4 = 16
2**5 = 32
```

```
In [86]: for i in range(10):
             print('2**{} = {}'.format(i, 2 ** i))

2**0 = 1
2**1 = 2
2**2 = 4
2**3 = 8
2**4 = 16
2**5 = 32
2**6 = 64
2**7 = 128
2**8 = 256
2**9 = 512
```

```
In [87]: # there is no increment/decrement operator ++/--!
         i = 0
         while i < 10:
             print('2**{} = {}'.format(i, 2 ** i))
             i += 1

2**0 = 1
2**1 = 2
2**2 = 4
2**3 = 8
2**4 = 16
2**5 = 32
2**6 = 64
2**7 = 128
2**8 = 256
2**9 = 512
```

An often used pattern is to iterate over a sequence type.

```
In [88]: L = ['lion', 'leopard', 'rhinoceros', 'elephant', 'buffalo']
         L

Out[88]: ['lion', 'leopard', 'rhinoceros', 'elephant', 'buffalo']

In [89]: for animal in L:
             print(animal)

lion
leopard
rhinoceros
elephant
buffalo


In [90]: for i in range(len(L)):
             print('{:02d}: {}'.format(i, L[i]))

00: lion
01: leopard
02: rhinoceros
03: elephant
04: buffalo


In [91]: # python provides tuple unpacking
         t = ('a', 'b')
         x, y = t
         print('x: {} y: {}'.format(x, y))

x: a y: b


In [92]: for i, animal in enumerate(L): # enumerate creates a list of tuples to iterate on
             print('{:02d}: {}'.format(i, L[i]))

00: lion
01: leopard
02: rhinoceros
03: elephant
04: buffalo
```

Python also provides the classical `break` and `continue` statements. A special statement is `pass` which does nothing but is required for statements after a `:`

```
In [93]: animal = 'rhinoceros'
         if animal == 'lion':
             print("It's still a cat, but a large one!")
         else:
             pass
```

1.7 3.02 Functions

Functions can be defined using a lambda expression or via `def`. Python provides for functions both positional and keyword-based arguments.

```
In [94]: square = lambda x: x * x

In [95]: square(10)

Out[95]: 100

In [96]: # roots of ax^2 + bx + c
         quadratic_root = lambda a, b, c: ((-b - (b * b - 4 * a * c) ** .5) / (2 * a), (-b + (

In [97]: quadratic_root(1, 5.5, -10.5)

Out[97]: (-7.0, 1.5)

In [98]: # a clearer function using def
         def quadratic_root(a, b, c):
             d = (b * b - 4 * a * c) ** .5

             coeff = .5 / a

             return (coeff * (-b - d), coeff * (-b + d))

In [99]: quadratic_root(1, 5.5, -10.5)

Out[99]: (-7.0, 1.5)
```

Functions can have positional arguments and keyword based arguments. Positional arguments have to be declared before keyword args

```
In [100]: # name is a positional argument, message a keyword argument
          def greet(name, message='Hello {}, how are you today?'):
              print(message.format(name))

In [101]: greet('Tux')

Hello Tux, how are you today?


In [102]: greet('Tux', 'Hi {}!')

Hi Tux!


In [103]: greet('Tux', message='What\'s up {}?')

What's up Tux?
```

```
In [104]: # this doesn't work
          greet(message="Hi {} !", 'Tux')


          File "<ipython-input-104-0f79efc3a31e>", line 2
        greet(message="Hi {} !", 'Tux')
                                 ^
    SyntaxError: positional argument follows keyword argument
```

keyword arguments can be used to define default values

```
In [105]: import math

          def log(num, base=math.e):
              return math.log(num) / math.log(base)

In [106]: log(math.e)

Out[106]: 1.0

In [107]: log(10)

Out[107]: 2.302585092994046

In [108]: log(1000, 10)

Out[108]: 2.9999999999999996
```

1.8 3.03 builtin functions, attributes

Python provides a rich standard library with many builtin functions. Also, bools/ints/floats/strings have many builtin methods allowing for concise code.

One of the most useful builtin function is `help`. Call it on any object to get more information, what methods it supports.

```
In [109]: s = 'sealion'

In [110]: help(str) # or help(type(s))

Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
```

```
|  Otherwise, returns the result of object.__str__() (if defined)
|  or repr(object).
|  encoding defaults to sys.getdefaultencoding().
|  errors defaults to 'strict'.
|
|  Methods defined here:
|
|  __add__(self, value, /)
|      Return self+value.
|
|  __contains__(self, key, /)
|      Return key in self.
|
|  __eq__(self, value, /)
|      Return self==value.
|
|  __format__(self, format_spec, /)
|      Return a formatted version of the string as described by format_spec.
|
|  __ge__(self, value, /)
|      Return self>=value.
|
|  __getattribute__(self, name, /)
|      Return getattr(self, name).
|
|  __getitem__(self, key, /)
|      Return self[key].
|
|  __getnewargs__(...)
|
|  __gt__(self, value, /)
|      Return self>value.
|
|  __hash__(self, /)
|      Return hash(self).
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
```

```
|  __mod__(self, value, /)
|      Return self%value.
|
|  __mul__(self, value, /)
|      Return self*value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __rmod__(self, value, /)
|      Return value%self.
|
|  __rmul__(self, value, /)
|      Return value*self.
|
|  __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.
|
|  center(self, width, fillchar=' ', /)
|      Return a centered string of length width.
|
|      Padding is done using the specified fill character (default is a space).
|
|  count(...)
|      S.count(sub[, start[, end]]) -> int
|
|      Return the number of non-overlapping occurrences of substring sub in
|      string S[start:end].  Optional arguments start and end are
|      interpreted as in slice notation.
|
|  encode(self, /, encoding='utf-8', errors='strict')
|      Encode the string using the codec registered for encoding.
|
```

```
|      encoding
|        The encoding in which to encode the string.
|      errors
|        The error handling scheme to use for encoding errors.
|        The default is 'strict' meaning that encoding errors raise a
|        UnicodeEncodeError.  Other possible values are 'ignore', 'replace' and
|        'xmlcharrefreplace' as well as any other name registered with
|        codecs.register_error that can handle UnicodeEncodeErrors.
|
|  endswith(...)
|      S.endswith(suffix[, start[, end]]) -> bool
|
|      Return True if S ends with the specified suffix, False otherwise.
|      With optional start, test S beginning at that position.
|      With optional end, stop comparing S at that position.
|      suffix can also be a tuple of strings to try.
|
|  expandtabs(self, /, tabsize=8)
|      Return a copy where all tab characters are expanded using spaces.
|
|      If tabsize is not given, a tab size of 8 characters is assumed.
|
|  find(...)
|      S.find(sub[, start[, end]]) -> int
|
|      Return the lowest index in S where substring sub is found,
|      such that sub is contained within S[start:end].  Optional
|      arguments start and end are interpreted as in slice notation.
|
|      Return -1 on failure.
|
|  format(...)
|      S.format(*args, **kwargs) -> str
|
|      Return a formatted version of S, using substitutions from args and kwargs.
|      The substitutions are identified by braces ('{' and '}').
|
|  format_map(...)
|      S.format_map(mapping) -> str
|
|      Return a formatted version of S, using substitutions from mapping.
|      The substitutions are identified by braces ('{' and '}').
|
|  index(...)
|      S.index(sub[, start[, end]]) -> int
|
|      Return the lowest index in S where substring sub is found,
|      such that sub is contained within S[start:end].  Optional
```

```
|      arguments start and end are interpreted as in slice notation.
|
|      Raises ValueError when the substring is not found.
|
|  isalnum(self, /)
|      Return True if the string is an alpha-numeric string, False otherwise.
|
|      A string is alpha-numeric if all characters in the string are alpha-numeric and
|      there is at least one character in the string.
|
|  isalpha(self, /)
|      Return True if the string is an alphabetic string, False otherwise.
|
|      A string is alphabetic if all characters in the string are alphabetic and there
|      is at least one character in the string.
|
|  isascii(self, /)
|      Return True if all characters in the string are ASCII, False otherwise.
|
|      ASCII characters have code points in the range U+0000-U+007F.
|      Empty string is ASCII too.
|
|  isdecimal(self, /)
|      Return True if the string is a decimal string, False otherwise.
|
|      A string is a decimal string if all characters in the string are decimal and
|      there is at least one character in the string.
|
|  isdigit(self, /)
|      Return True if the string is a digit string, False otherwise.
|
|      A string is a digit string if all characters in the string are digits and there
|      is at least one character in the string.
|
|  isidentifier(self, /)
|      Return True if the string is a valid Python identifier, False otherwise.
|
|      Use keyword.iskeyword() to test for reserved identifiers such as "def" and
|      "class".
|
|  islower(self, /)
|      Return True if the string is a lowercase string, False otherwise.
|
|      A string is lowercase if all cased characters in the string are lowercase and
|      there is at least one cased character in the string.
|
|  isnumeric(self, /)
|      Return True if the string is a numeric string, False otherwise.
```

```
|
|      A string is numeric if all characters in the string are numeric and there is at
|      least one character in the string.
|
|  isprintable(self, /)
|      Return True if the string is printable, False otherwise.
|
|      A string is printable if all of its characters are considered printable in
|      repr() or if it is empty.
|
|  isspace(self, /)
|      Return True if the string is a whitespace string, False otherwise.
|
|      A string is whitespace if all characters in the string are whitespace and there
|      is at least one character in the string.
|
|  istitle(self, /)
|      Return True if the string is a title-cased string, False otherwise.
|
|      In a title-cased string, upper- and title-case characters may only
|      follow uncased characters and lowercase characters only cased ones.
|
|  isupper(self, /)
|      Return True if the string is an uppercase string, False otherwise.
|
|      A string is uppercase if all cased characters in the string are uppercase and
|      there is at least one cased character in the string.
|
|  join(self, iterable, /)
|      Concatenate any number of strings.
|
|      The string whose method is called is inserted in between each given string.
|      The result is returned as a new string.
|
|      Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'
|
|  ljust(self, width, fillchar=' ', /)
|      Return a left-justified string of length width.
|
|      Padding is done using the specified fill character (default is a space).
|
|  lower(self, /)
|      Return a copy of the string converted to lowercase.
|
|  lstrip(self, chars=None, /)
|      Return a copy of the string with leading whitespace removed.
|
|      If chars is given and not None, remove characters in chars instead.
```

```
|
|  partition(self, sep, /)
|      Partition the string into three parts using the given separator.
|
|      This will search for the separator in the string.  If the separator is found,
|      returns a 3-tuple containing the part before the separator, the separator
|      itself, and the part after it.
|
|      If the separator is not found, returns a 3-tuple containing the original string
|      and two empty strings.
|
|  replace(self, old, new, count=-1, /)
|      Return a copy with all occurrences of substring old replaced by new.
|
|        count
|          Maximum number of occurrences to replace.
|          -1 (the default value) means replace all occurrences.
|
|      If the optional argument count is given, only the first count occurrences are
|      replaced.
|
|  rfind(...)
|      S.rfind(sub[, start[, end]]) -> int
|
|      Return the highest index in S where substring sub is found,
|      such that sub is contained within S[start:end].  Optional
|      arguments start and end are interpreted as in slice notation.
|
|      Return -1 on failure.
|
|  rindex(...)
|      S.rindex(sub[, start[, end]]) -> int
|
|      Return the highest index in S where substring sub is found,
|      such that sub is contained within S[start:end].  Optional
|      arguments start and end are interpreted as in slice notation.
|
|      Raises ValueError when the substring is not found.
|
|  rjust(self, width, fillchar=' ', /)
|      Return a right-justified string of length width.
|
|      Padding is done using the specified fill character (default is a space).
|
|  rpartition(self, sep, /)
|      Partition the string into three parts using the given separator.
|
|      This will search for the separator in the string, starting at the end. If
```

```
|      the separator is found, returns a 3-tuple containing the part before the
|      separator, the separator itself, and the part after it.
|
|      If the separator is not found, returns a 3-tuple containing two empty strings
|      and the original string.
|
|  rsplit(self, /, sep=None, maxsplit=-1)
|      Return a list of the words in the string, using sep as the delimiter string.
|
|        sep
|          The delimiter according which to split the string.
|          None (the default value) means split according to any whitespace,
|          and discard empty strings from the result.
|        maxsplit
|          Maximum number of splits to do.
|          -1 (the default value) means no limit.
|
|      Splits are done starting at the end of the string and working to the front.
|
|  rstrip(self, chars=None, /)
|      Return a copy of the string with trailing whitespace removed.
|
|      If chars is given and not None, remove characters in chars instead.
|
|  split(self, /, sep=None, maxsplit=-1)
|      Return a list of the words in the string, using sep as the delimiter string.
|
|      sep
|        The delimiter according which to split the string.
|        None (the default value) means split according to any whitespace,
|        and discard empty strings from the result.
|      maxsplit
|        Maximum number of splits to do.
|        -1 (the default value) means no limit.
|
|  splitlines(self, /, keepends=False)
|      Return a list of the lines in the string, breaking at line boundaries.
|
|      Line breaks are not included in the resulting list unless keepends is given and
|      true.
|
|  startswith(...)
|      S.startswith(prefix[, start[, end]]) -> bool
|
|      Return True if S starts with the specified prefix, False otherwise.
|      With optional start, test S beginning at that position.
|      With optional end, stop comparing S at that position.
|      prefix can also be a tuple of strings to try.
```

```
|
|  strip(self, chars=None, /)
|      Return a copy of the string with leading and trailing whitespace remove.
|
|      If chars is given and not None, remove characters in chars instead.
|
|  swapcase(self, /)
|      Convert uppercase characters to lowercase and lowercase characters to uppercase.
|
|  title(self, /)
|      Return a version of the string where each word is titlecased.
|
|      More specifically, words start with uppercased characters and all remaining
|      cased characters have lower case.
|
|  translate(self, table, /)
|      Replace each character in the string using the given translation table.
|
|        table
|          Translation table, which must be a mapping of Unicode ordinals to
|          Unicode ordinals, strings, or None.
|
|      The table must implement lookup/indexing via __getitem__, for instance a
|      dictionary or list.  If this operation raises LookupError, the character is
|      left untouched.  Characters mapped to None are deleted.
|
|  upper(self, /)
|      Return a copy of the string converted to uppercase.
|
|  zfill(self, width, /)
|      Pad a numeric string with zeros on the left, to fill a field of the given width.
|
|      The string is never truncated.
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  maketrans(x, y=None, z=None, /)
|      Return a translation table usable for str.translate().
|
|      If there is only one argument, it must be a dictionary mapping Unicode
|      ordinals (integers) or characters to Unicode ordinals, strings or None.
|      Character keys will be then converted to ordinals.
|      If there are two arguments, they must be strings of equal length, and
|      in the resulting dictionary, each character in x will be mapped to the
```

```
 |      character at the same position in y. If there is a third argument, it
 |      must be a string, whose characters will be mapped to None in the result.
```

```
In [111]: s.capitalize()
```

```
Out[111]: 'Sealion'
```

```
In [112]: help(int)
```

```
Help on class int in module builtins:

class int(object)
 |  int([x]) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
 |
 |  If x is not a number or if base is given, then x must be a string,
 |  bytes, or bytearray instance representing an integer literal in the
 |  given base.  The literal can be preceded by '+' or '-' and be surrounded
 |  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
 |  Base 0 means to interpret the base from the string as an integer literal.
 |  >>> int('0b100', base=0)
 |  4
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __and__(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 lis
|
|  __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
|
|  __or__(self, value, /)
|      Return self|value.
|
|  __pos__(self, /)
|      +self
|
|  __pow__(self, value, mod=None, /)
|      Return pow(self, value, mod).
|
|  __radd__(self, value, /)
|      Return value+self.
|
|  __rand__(self, value, /)
|      Return value&self.
|
|  __rdivmod__(self, value, /)
|      Return divmod(value, self).
|
|  __repr__(self, /)
|      Return repr(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.
|
|  __ror__(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.
|
|  __rxor__(self, value, /)
|      Return value^self.
|
|  __sizeof__(self, /)
|      Returns size in memory, in bytes.
|
|  __str__(self, /)
|      Return str(self).
|
|  __sub__(self, value, /)
|      Return self-value.
|
|  __truediv__(self, value, /)
|      Return self/value.
|
|  __trunc__(...)
|      Truncating an Integral returns itself.
|
|  __xor__(self, value, /)
|      Return self^value.
|
|  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 defined here:
|
|  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.
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
```

```
 |  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 [113]: x = int('-10')
          x

Out[113]: -10
```

For casting objects, python provides several functions closely related to the constructors `bool`, `int`, `float`, `str`, `list`, `tuple`, `dict`, ...

```
In [114]: tuple([1, 2, 3, 4])

Out[114]: (1, 2, 3, 4)

In [115]: str((1, 2, 3))

Out[115]: '(1, 2, 3)'

In [116]: str([1, 4.5])

Out[116]: '[1, 4.5]'
```

1.9 4.01 Dictionaries

Dictionaries (or associate arrays) provide a structure to lookup values based on keys. I.e. they're a collection of k->v pairs.

```
In [117]: list(zip(['brand', 'model', 'year'], ['Ford', 'Mustang', 1964])) # creates a list of

Out[117]: [('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)]

In [118]: # convert a list of tuples to a dictionary
          D = dict(zip(['brand', 'model', 'year'], ['Ford', 'Mustang', 1964]))
          D

Out[118]: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}

In [119]: D['brand']
```

```
Out[119]: 'Ford'

In [120]: D = dict([('brand', 'Ford'), ('model', 'Mustang')])
          D['model']

Out[120]: 'Mustang'
```

Dictionaries can be also directly defined using `{ ... : ..., ...}` syntax

```
In [121]: D = {'brand' : 'Ford', 'model' : 'Mustang', 'year' : 1964}

In [122]: D

Out[122]: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}

In [123]: # dictionaries have serval useful functions implemented
          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
```

```
In [124]: # adding a new key
          D['price'] = '48k'

In [125]: D

Out[125]: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'price': '48k'}

In [126]: # removing a key
          del D['year']
```

```
In [127]: D

Out[127]: {'brand': 'Ford', 'model': 'Mustang', 'price': '48k'}

In [128]: # checking whether a key exists
          'brand' in D

Out[128]: True

In [129]: # returning a list of keys
          D.keys()

Out[129]: dict_keys(['brand', 'model', 'price'])

In [130]: # casting to a list
          list(D.keys())

Out[130]: ['brand', 'model', 'price']

In [131]: D

Out[131]: {'brand': 'Ford', 'model': 'Mustang', 'price': '48k'}

In [132]: # iterating over a dictionary
          for k in D.keys():
              print(k)

brand
model
price


In [133]: for v in D.values():
              print(v)

Ford
Mustang
48k


In [134]: for k, v in D.items():
              print('{}: {}'.format(k, v))

brand: Ford
model: Mustang
price: 48k
```

1.10 4.02 Calling functions with tuples/dicts

Python provides two special operators * and ** to call functions with arguments specified through a tuple or dictionary. I.e. * unpacks a tuple into positional args, whereas ** unpacks a dictionary into keyword arguments.

```
In [135]: quadratic_root(1, 5.5, -10.5)

Out[135]: (-7.0, 1.5)

In [136]: args=(1, 5.5, -10.5)
          quadratic_root(*args)

Out[136]: (-7.0, 1.5)

In [137]: args=('Tux',) # to create a tuple with one element, need to append , !
          kwargs={'message' : 'Hi {}!'}
          greet(*args, **kwargs)

Hi Tux!
```

1.11 5.01 Basic I/O

Python has builtin support to handle files

```
In [138]: f = open('file.txt', 'w')

          f.write('Hello world')
          f.close()
```

Because a file needs to be closed (i.e. the file object destructed), python has a handy statement to deal with auto-closing/destruction: The `with` statement.

```
In [139]: with open('file.txt', 'r') as f:
              lines = f.readlines()
              print(lines)

['Hello world']
```

Again, `help` is useful to understand what methods a file object has

```
In [140]: help(f)

Help on TextIOWrapper object:

class TextIOWrapper(_TextIOBase)
 |  TextIOWrapper(buffer, encoding=None, errors=None, newline=None, line_buffering=False, write
 |
 |  Character and line based layer over a BufferedIOBase object, buffer.
```

```
|
|  encoding gives the name of the encoding that the stream will be
|  decoded or encoded with. It defaults to locale.getpreferredencoding(False).
|
|  errors determines the strictness of encoding and decoding (see
|  help(codecs.Codec) or the documentation for codecs.register) and
|  defaults to "strict".
|
|  newline controls how line endings are handled. It can be None, '',
|  '\n', '\r', and '\r\n'.  It works as follows:
|
|  * On input, if newline is None, universal newlines mode is
|    enabled. Lines in the input can end in '\n', '\r', or '\r\n', and
|    these are translated into '\n' before being returned to the
|    caller. If it is '', universal newline mode is enabled, but line
|    endings are returned to the caller untranslated. If it has any of
|    the other legal values, input lines are only terminated by the given
|    string, and the line ending is returned to the caller untranslated.
|
|  * On output, if newline is None, any '\n' characters written are
|    translated to the system default line separator, os.linesep. If
|    newline is '' or '\n', no translation takes place. If newline is any
|    of the other legal values, any '\n' characters written are translated
|    to the given string.
|
|  If line_buffering is True, a call to flush is implied when a call to
|  write contains a newline character.
|
|  Method resolution order:
|      TextIOWrapper
|      _TextIOBase
|      _IOBase
|      builtins.object
|
|  Methods defined here:
|
|  __getstate__(...)
|
|  __init__(self, /, *args, **kwargs)
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  __next__(self, /)
|      Implement next(self).
|
|  __repr__(self, /)
|      Return repr(self).
|
|  close(self, /)
```

```
|      Flush and close the IO object.
|
|      This method has no effect if the file is already closed.
|
|  detach(self, /)
|      Separate the underlying buffer from the TextIOBase and return it.
|
|      After the underlying buffer has been detached, the TextIO is in an
|      unusable state.
|
|  fileno(self, /)
|      Returns underlying file descriptor if one exists.
|
|      OSError is raised if the IO object does not use a file descriptor.
|
|  flush(self, /)
|      Flush write buffers, if applicable.
|
|      This is not implemented for read-only and non-blocking streams.
|
|  isatty(self, /)
|      Return whether this is an 'interactive' stream.
|
|      Return False if it can't be determined.
|
|  read(self, size=-1, /)
|      Read at most n characters from stream.
|
|      Read from underlying buffer until we have n characters or we hit EOF.
|      If n is negative or omitted, read until EOF.
|
|  readable(self, /)
|      Return whether object was opened for reading.
|
|      If False, read() will raise OSError.
|
|  readline(self, size=-1, /)
|      Read until newline or EOF.
|
|      Returns an empty string if EOF is hit immediately.
|
|  reconfigure(self, /, *, encoding=None, errors=None, newline=None, line_buffering=None, writ
|      Reconfigure the text stream with new parameters.
|
|      This also does an implicit stream flush.
|
|  seek(self, cookie, whence=0, /)
|      Change stream position.
```

```
|
|      Change the stream position to the given byte offset. The offset is
|      interpreted relative to the position indicated by whence.  Values
|      for whence are:
|
|      * 0 -- start of stream (the default); offset should be zero or positive
|      * 1 -- current stream position; offset may be negative
|      * 2 -- end of stream; offset is usually negative
|
|      Return the new absolute position.
|
|  seekable(self, /)
|      Return whether object supports random access.
|
|      If False, seek(), tell() and truncate() will raise OSError.
|      This method may need to do a test seek().
|
|  tell(self, /)
|      Return current stream position.
|
|  truncate(self, pos=None, /)
|      Truncate file to size bytes.
|
|      File pointer is left unchanged.  Size defaults to the current IO
|      position as reported by tell().  Returns the new size.
|
|  writable(self, /)
|      Return whether object was opened for writing.
|
|      If False, write() will raise OSError.
|
|  write(self, text, /)
|      Write string to stream.
|      Returns the number of characters written (which is always equal to
|      the length of the string).
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  buffer
|
|  closed
```

```
|
|  encoding
|      Encoding of the text stream.
|
|      Subclasses should override.
|
|  errors
|      The error setting of the decoder or encoder.
|
|      Subclasses should override.
|
|  line_buffering
|
|  name
|
|  newlines
|      Line endings translated so far.
|
|      Only line endings translated during reading are considered.
|
|      Subclasses should override.
|
|  write_through
|
|  ----------------------------------------------------------------------
|  Methods inherited from _IOBase:
|
|  __del__(...)
|
|  __enter__(...)
|
|  __exit__(...)
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  readlines(self, hint=-1, /)
|      Return a list of lines from the stream.
|
|      hint can be specified to control the number of lines read: no more
|      lines will be read if the total size (in bytes/characters) of all
|      lines so far exceeds hint.
|
|  writelines(self, lines, /)
|      Write a list of lines to stream.
|
|      Line separators are not added, so it is usual for each of the
|      lines provided to have a line separator at the end.
```

```
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from _IOBase:
|
|  __dict__
```

End of lecture