Floats, Unicode, Regular Expressions

Floating Point (float)

- Use a decimal or exponential notation

```
a = 37.45
b = 4e5
c = -1.345e-10
```

- Represented as "double precision" (64bit) using the native CPU representation, following the IEEE 754 spec

```
17 digits of precision
Exponent from -308 to 308
```

- The same as the C double type (and in most other languages)

Floating Point

- Beware that floating point numbers are inexact

```
>>> a = 2.1 + 4.2
>>> a == 6.3
False
>>> a
6.300000000000001
```

- This is not specific to Python, it's how floating point numbers work
- Floats only have 17 bits of precision
- The results of calculations may not be exactly what you expect (not a Python bug!)

Floating Point

- Additional functions are in the math module

```
import math
a = math.sqrt(x)
b = math.sin(x)
c = math.cos(x)
d = math.tan(x)
e = math.log(x)
```

- Note: floats have two zeroes! `+0.0` and `-0.0`

- Infinity and NaN (Not A Number) exist:

```
>>> float('inf') / float('inf')
nan
>>> nan = float('nan')
>>> nan == nan
False
```

Comparing Floats

- Use `math.isclose()` to compare floats:

```
>>> a = 4.2 + 2.1
>>> math.isclose(a, 6.3)
True
>>> help(math.isclose)
Help on built-in function isclose in module math:

isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)
    Determine whether two floating point numbers are close in
value.
```

- In tests (see also `unittest.TestCase.assertAlmostEquals`):

```
>>> import pytest
>>> assert pytest.approx(a) == 6.3
>>> assert a == pytest.approx(6.3, 0.0000001)
```

Text Processing

Strings, Unicode and bytes
Regular Expressions

The String Type

- Python strings are Unicode and the type is “str”
- Strings are immutable
- Strings are sequences – they can be iterated over, sliced and concatenated
- Many (many) useful methods
- String formatting with f-strings
- String templating with the format methods
- Binary data is supported through “bytes”

Strings

- Backslash is the escape character (\t, \n, \', \", \\, ...)
- Newlines are always "\n" inside Python
- Single and double quotes are equivalent
- Triple quotes for multiline strings
- Use the 'r' prefix for 'raw' strings (regular expressions and Windows paths)

```
>>> """This is a
... multiline
... string"""
'This is a\nmultiline\nstring'

>>> r'c:\Users\fred'
'c:\\Users\\fred'

>>> print("\t means tab")
    means tab
```

Strings as Sequences

- Lists, tuples and strings are all sequences
- Iteration over strings is by character (not so useful)
- Strings can be indexed into : s[n]

```
s = 'Hello world'
s[0] == 'H'
s[-1] == 'd'
```

- Slicing (for structured data) : s[start:end]

```
s[2:4] = 'll'
s[6:] = 'world'
s[:5] = 'Hello'
```

- Strings can be added (concatenated)

```
'Hello' + ' ' + 'world' == 'Hello world'
```

String Methods

```
>>> s = 'Hello world    '
>>> s.strip()
'Hello world'
>>> s.upper()
'HELLO WORLD    '
>>> s.replace('world', 'London')
'Hello London    '
```

```
>>> [method for method in dir(s) if not method.startswith('_')]
['capitalize', 'casefold', 'center', 'count', 'encode', 'endswi
th', 'expandtabs', 'find', 'format', 'format_map', 'index', 'is
alnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidenti
fier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istit
le', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans'
, 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartiti
on', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', '
strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

- Strings are immutable, cannot be changed

```
>>> s = 'Hello World'
>>> s[0] = 'F'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

- All operations on strings return a *new* string
- Any object can be converted into a string with "str"

```
>>> number = 1_000_000
>>> str(number)
'1000000'
```

- Substrings can be found with the in operator

```
>>> 'Hello' in 'Hello world'
True
>>> 'London' not in 'Hello world'
True
```

String Formatting

- f-strings are evaluated immediately

```
>>> name = 'Michael'
>>> job = 'Python trainer'
>>> f'My name is {name} and my job is {job}'
'My name is Michael and my job is Python trainer'
```

- Formatting is controlled with the *fml* (formatting mini language), also used by the format methods

```
>>> colour = 'red'
>>> weight = 334.5601
>>> size = 'large'
>>> f'{colour:>10s} {weight:>10.2f} {size:>10s}'
'       red     334.56      large'
```

Format Methods

- The *format* methods provide reusable string templates

```
>>> template = '{colour:>10s} {weight:>10.2f} {size:>10s}'
>>> template.format(colour=colour, weight=weight, size=size)
'       red     334.56      large'

>>> template = '{:>10s} {:>10.2f} {:>10s}'
>>> template.format(colour, weight, size)
'       red     334.56      large'

>>> template = '{colour:>10s} {weight:>10.2f} {size:>10s}'
>>> data = {'colour': 'blue', 'weight': 478.998, 'size':
'huge'}
>>> template.format_map(data) # or template.format(**data)
'      blue     479.00       huge'
```

Formatting Mini Language

{name: pad-char alignment pad-amount type}

- Alignment (defaults to right):
  "<": left aligned
  ">": right aligned
  "^": centred
- Padding is an optional pad character and an amount

```
>>> f'{name:*^20s}'
'******Michael*******'
```

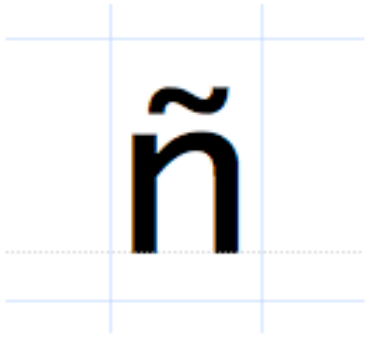
- Type codes: s for string, d for integer, f for float (with number of decimal places), plus many more less useful

```
>>> number = 34.921345
>>> f'{number:.2f}'
'34.92'
```

See: https://learnpython.com/blog/python-string-formatting/

Unicode

Internally strings are a sequence of Unicode code points.

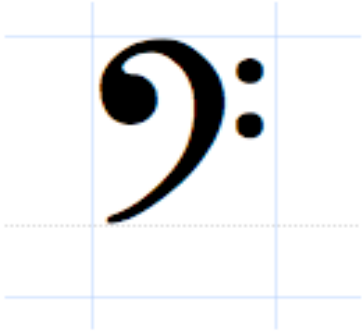


LATIN SMALL LETTER N WITH TILDE

Unicode U+00F1

FOR ALL

Unicode U+2200



MUSICAL SYMBOL F CLEF

Unicode U+1D122 (U+D834 U+DD22)

```
a = '\xf1'          # a = 'ñ'
b = '\u2200'        # b = '∀'
c = '\U0001D122'    # c = '𝄢'
d = '\N{FOR ALL}'   # d = '∀'
```

Binary Data with bytes

- The bytes type uses the 'b' string prefix
- The old 'str' type from Python 2
- Immutable, use bytearray for mutable data
- Only supports the old "%" style string formatting
- We can encode from text to bytes or decode from bytes to a Unicode string

```
>>> b'This is binary data \x00\x2f\x44'
b'This is binary data \x00/D'
>>> "This costs £456".encode('utf-8')
b'This costs \xc2\xa3456'
>>> type(b'some data')
<class 'bytes'>
```

Strings in the Standard Library

- There are several standard library modules for working with strings and text
  - string
  - textwrap
  - difflib
  - etc...

For more advanced use-cases use a proper templating engine (e.g. Jinja2) or a proper parser (e.g. PLY).

Regular Expressions

- A powerful mini language
- Used by most programming languages and many tools
- Originating in Perl, Python has its own dialect
- Useful for processing semi-structured text (or bytes)
- Provided by the re module
- Always use raw strings (r'..') for regular expressions!

- We'll look at:
  - Features
  - Readability
  - Common pitfalls

Anchors and Selection

- Character classes select characters to match

[abc] any one character from list (a OR b OR c)
[a-m] any one character from range (a OR b OR … OR m)
[^abc] any one character not in list (NOT a NOR b NOR c)
. any one character
\s any white space character
\S any NON white space character
\d any digit
\D any NON digit
\w any alphanumeric
\W any NON alphanumeric

- Anchors indicate context

^ start of string
$ end of string
\b on a word boundary
\B NOT on a word boundary

Repeat Counts

repeat counts apply to the previous expression

*	0 or more repeats
+	1 or more repeats
?	0 or 1 repeats
{n,m}	n to m repeats
{n,}	at least n repeats
{n}	exactly n repeats

Examples

[aeiou]*	0 more vowels
[0-9]+	1 or more digits (\d+)
[0-9]{3,5}	3-5 digits
\.?	0 or 1 periods

\. matches a literal period. The backslash escapes the special meaning of the "." (or any other special character) that follows

Pattern Matching Functions

- match : matches from the start of the text
- search : match anywhere in the text

```
>>> import re
>>> test = '--------ABC---------'
>>> pattern = r"\w+"
>>> print(re.match(pattern, test))
None
>>> re.search(pattern, test)
<re.Match object; span=(8, 11), match='ABC'>
>>> match = _
>>> match.group()
'ABC'
```

None is returned when there is no match.

The Walrus Operator

- New in Python 3.8, := is assignment as an expression
- Called "the walrus operator" it allows us to do an assignment and test in one operation
- Ideal for checking if a regular expression returns a match

```
>>> test = '--------ABC---------'
>>> pattern = r"\w+"
>>> if match := re.search(pattern, test):
...     print(f'We found a match {match!r}')
...
We found a match <re.Match object; span=(8, 11),
match='ABC'>
```

Pattern Matching Functions

Note: in the re functions the regex pattern is always the first argument followed by the text to match on (for the two argument functions).

There's also an optional, but obsolete, compile step. Internally compiled regular expressions are cached by the re module so compilation is no longer needed.

```
>>> test = '--------ABC---------'
>>> pattern = re.compile(r"\w+")
>>> pattern
re.compile('\\w+')
>>> pattern.search(test)
<re.Match object; span=(8, 11), match='ABC'>
>>> match = pattern.search(test)
>>> match.group()
'ABC'
```

Splitting Strings

Strings have a useful split method, but what if the divider is not always the same, extraneous whitespace for example? We can split on a regular expression.

```
>>> text = "aaa ; bbb  ;ccc; ddd ;  eee"
>>> pattern = r"\s*;\s*"
>>> re.split(pattern, text)
['aaa', 'bbb', 'ccc', 'ddd', 'eee']
```

Combining and Substitutions

- The syntax for matching alternatives is "|"
- The "sub" function does substitutions

```
>>> text = "red or blue or green"
>>> pattern = r"red|blue|green"
>>> re.sub(pattern, 'colour', text)
'colour or colour or colour'
```

Note: the "sub" function can take a replacer function instead of a string for more advanced substitution. See the documentation.

Flags

- Matching behaviour can be customised with flags
- Flags can be combined with "|"
- The most useful flags are:

re.IGNORECASE, re.I	ignore case when matching
re.MULTILINE, re.M	match across newlines
re.DOTALL, re.S	"." will match newlines
re.VERBOSE, re.X	allow comments in regular expressions

e.g. re.match(pattern, text, re.VERBOSE | re.IGNORECASE)

Comments

Making regular expressions more readable with comments and the re.VERBOSE flag.

```
>>> text = "AAAAAA1111BBB2222CCC33DDDDDD"
>>> pattern = r"""
...   ^     # start of line
...   (.*?) # 0 or more characters
...         # non-greedy
...   (\d+) # 1 or more digits
...   (.*)  # 0 or more characters
...   $     # end of line"""
>>> match = re.match(pattern, text, re.VERBOSE | re.IGNORECASE)
>>> match.group()
'AAAAAA1111BBB2222CCC33DDDDDD'
```

Repeating Patterns

For finding multiple matches use findall and finditer. finditer can be more memory efficient.

- findall : "greedy", returns a list
- finditer : "lazy", returns an iterator

```
>>> text = 'AB12CD34EF56GH'
>>> pattern = r"(\d+)"
>>> re.findall(pattern, text)
['12', '34', '56']
>>> re.finditer(pattern, text)
<callable_iterator object at 0x7fce06532a60>
>>> for match in re.finditer(pattern, text):
...     print(match.group(), end =', ')
...
12, 34, 56
```

Advanced Searches: Groups

Groups, with (), allow you to capture and extract parts of the text. Much of the power of regexes are in groups.

```
>>> text = '----111122333344445567777----'
>>> pattern = r'2+(3+)4+(5+)6+'
>>> match = re.search(pattern, text)
>>> match.groups()
('3333', '55')
>>> match.group(0)      # whole match, same as match.group()
'2233334444556'
>>> match.group(1)
'3333'
>>> match.group(2)
'55'
```

Note: you can use (?:) to use parentheses for clarity without creating a group.

Modifying Data with Groups

We can use groups and the "sub" function to modify and rewrite our text data. We refer to groups by number (or name which is more verbose).

```
>>> text = "This course is attended by Sarah and Aisha"
>>> pattern = r"^This course is attended by (.+) and (.+)$"
>>> replace = r"This course is attended by \2 and \1."
>>> re.sub(pattern, replace, text)
'This course is attended by Aisha and Sarah.'
```

Named groups allow dictionary access to groups:

```
>>> text = "This course is attended by Sarah and Aisha"
>>> pattern = r"^.* (?P<name1>\w+) and (?P<name2>\w+)$"
>>> match = re.match(pattern, text)
>>> match.groupdict()
{'name1': 'Sarah', 'name2': 'Aisha'}
```

Greedy versus non-Greedy

Here be dragons!

- Regular expressions work by "backtracking" and are "greedy", capturing as much as possible
- Adding a "?" to a pattern makes it non-greedy and here gets the match we want

```
>>> text = "AAAA1111BBBB2222CCCC3333DDDD"
>>> greedy = r"^(.+)(\d+)(.+)$"
>>> re.match(greedy, text).groups()
('AAAA1111BBBB2222CCCC333', '3', 'DDDD')

>>> non_greedy = r"^(.+?)(\d+)(.+)$"
>>> re.match(non_greedy, text).groups()
('AAAA', '1111', 'BBBB2222CCCC3333DDDD')
```

More Python

- A few very useful bits of Python syntax we haven't already covered
- Some of them from more recent versions of Python
  - Generator expressions
  - Ordered dictionaries
  - Ternary expressions
  - The walrus operator
  - Positional and keyword only arguments

Generator Expressions

- List comprehensions are "eager", they consume their input and produce a list
- Many of the builtin functions in Python are "lazy", they produce iterable objects instead of executing immediately

```
>>> range(100)
range(0, 100)
>>> zip(['name', 'shares', 'prices'], ['GOOG', 100, 490.10])
<zip object at 0x7f503b5d80c0>
>>> enumerate(nums)
<enumerate object at 0x7f503b6a8840>
```

- Generator expressions are a lazy version of list comprehensions

Generator Expressions

- Generator expressions produce "one shot" generators
- The syntax is very similar to list comprehensions

```
>>> a = [2, 4, 6, 8, 10]
>>> b = (x**2 for x in a)
>>> b
<generator object <genexpr> at 0x7f503b78f760>
>>> for result in b:
...     print(result)
...
4
16
36
64
100
```

- They don't produce a list, so the whole result set doesn't need to be in memory
- They can't be reused

Generator Expressions

- General syntax (very similar to list comprehensions)

```
(expression for names in iterable if conditional)
```

- They look better than list comprehensions in function calls

```
sum(x*x for x in a)
```

- Can be applied to any iterable and even chained together

```
>>> a = [1,2,3,4]
>>> b = (x*x for x in a)
>>> c = (-x for x in b)
>>> for i in c:
...     print(i, end=' ')
...
-1 -4 -9 -16
```

Ordered Dictionaries

- Since Python 3.7 Python dictionaries are now ordered by insertion order
- Originally a memory (layout) optimisation originating in pypy and ported to CPython
- Iteration over dictionaries, the keys and values and items, all preserve order

```
>>> d = {}
>>> d['first'] = 1
>>> d['second'] = 2
>>> d['third'] = 3
>>> list(d)
['first', 'second', 'third']
>>> d.keys()
dict_keys(['first', 'second', 'third'])
>>> d.items()
dict_items([('first', 1), ('second', 2), ('third', 3)])
>>> d.values()
dict_values([1, 2, 3])
```

Ternary Expressions

- Also known as "conditional expressions"
- A concise way of having an expression evaluate to a value based on a condition

```
>>> email_address = None
>>> send_email = True if email_address is not None else
False
>>> send_email
False
>>> email_address = 'michael@python.org'
>>> send_email = True if email_address is not None else
False
>>> send_email
True
```

- Very terse syntax, it maybe clearer to use if/else
- Like list comprehensions it can be helpful to start reading them in the middle (the true condition is on the left, the false is on the right and the if is in the middle)

The Walrus Operator

- Assignment as an expression: `x := 3`
- New in Python 3.8
- Useful where you need to test a value immediately after setting it
- Can be used to write inscrutable code!
- Old code (regular expressions):

```
>>> match = re.match(r'\w+@(\w+\.\w+)', email_address)
>>> if match is not None:
...     domain = match.groups(1)
...
```

- With the walrus operator:

```
>>> if match := re.match(r'\w+@(\w+\.\w+)', email_address):
...     domain = match.groups(1)
...
```

Positional and Keyword Only Arguments

- Python function signatures are now very rich
- We can now express positional and keyword only arguments
- Positional only arguments (mostly for compatibility with C functions) added in Python 3.8
- Keyword only arguments were new in Python 3.0

```
>>> def foo(data, /, *, debug=False):
...     pass
...
>>> foo(1, debug=True)
>>> foo(data=2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() got some positional-only arguments passed
as keyword arguments: 'data'
>>> foo(3, False)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() takes 1 positional argument but 2 were
given
```