California State University Fullerton

CPSC-223P

Python Programming

Stephen T. May

Python Tutorial

Section 7

Input and Output

https://docs.python.org/release/3.9.6/tutorial/index.html

Slide Notes

- Command typed at the Linux command prompt ($)

```
$ python3.9
```

- Command typed at the Python interpreter command prompt (>>>)

```
>>> Ctrl-D
```

- Python source code

```
print(“Hello world!”)
```

- Mixed example

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

7.1. Fancier Output Formatting

- To use formatted string literals, begin a string with `f` or `F` before the opening quotation mark or triple quotation mark
- Inside this string, you can write a Python expression between `{` and `}` characters that can refer to variables or literal values
- The `str.format()` method of strings requires more manual effort.
- You'll still use `{` and `}` to mark where a variable will be substituted and can provide detailed formatting directives, but you'll also need to provide the information to be formatted.
- You can also do all the string handling yourself by using string slicing and concatenation operations to create any layout you can imagine
- The string type has some methods that perform useful operations for padding strings to a given column width

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

```
>>> yes_votes = 42_572_654
>>> no_votes = 43_132_495
>>> percentage = yes_votes / (yes_votes + no_votes)
>>> '{:-9} YES votes  {:2.2%}'.format(yes_votes, percentage)
' 42572654 YES votes  49.67%'
```

7.1. Fancier Output Formatting (cont.)

- The `str()` function is meant to return representations of values which are fairly human-readable
- The `repr()` function is meant to generate representations which can be read by the interpreter
- For objects which don't have a particular representation for human consumption, `str()` will return the same value as `repr()`
- Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function
- Strings have two distinct representations

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) +
'...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

7.1.1. Formatted String Literals

- *Formatted string literals* let you include the value of Python expressions inside a string by prefixing the string with `f` or `F` and writing expressions as `{expression}`.
- An optional format specifier can follow the expression
- This allows greater control over how the value is formatted
- Passing an integer after the `':'` will cause that field to be a minimum number of characters wide
- This is useful for making columns line up
- Other modifiers can be used to convert the value before it is formatted
- `'!a'` applies `ascii()`
- `'!s'` applies `str()`
- `'!r'` applies `repr()`

```
>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.
```

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print(f'{name:10} ==> {phone:10d}')
...
Sjoerd     ==>       4127
Jack       ==>       4098
Dcab       ==>       7678
```

```
>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.
```

7.1.1. Format Specification Mini-Language

https://docs.python.org/release/3.9.6/library/string.html#formatspec

```
format_spec     ::=  [[fill]align][sign][#][0][width][grouping_option][.precision][type]
fill            ::=  <any character>
align           ::=  "<" | ">" | "=" | "^"
sign            ::=  "+" | "-" | " "
width           ::=  digit+
grouping_option ::=  "_" | ","
precision       ::=  digit+
type            ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" | "x" | "X" | "%"
```

The available *string* presentation types are:

Type	Meaning
`'s'`	String format. This is the default type for strings and may be omitted.
None	The same as `'s'`.

7.1.1. Format Specification Mini-Language (cont.)

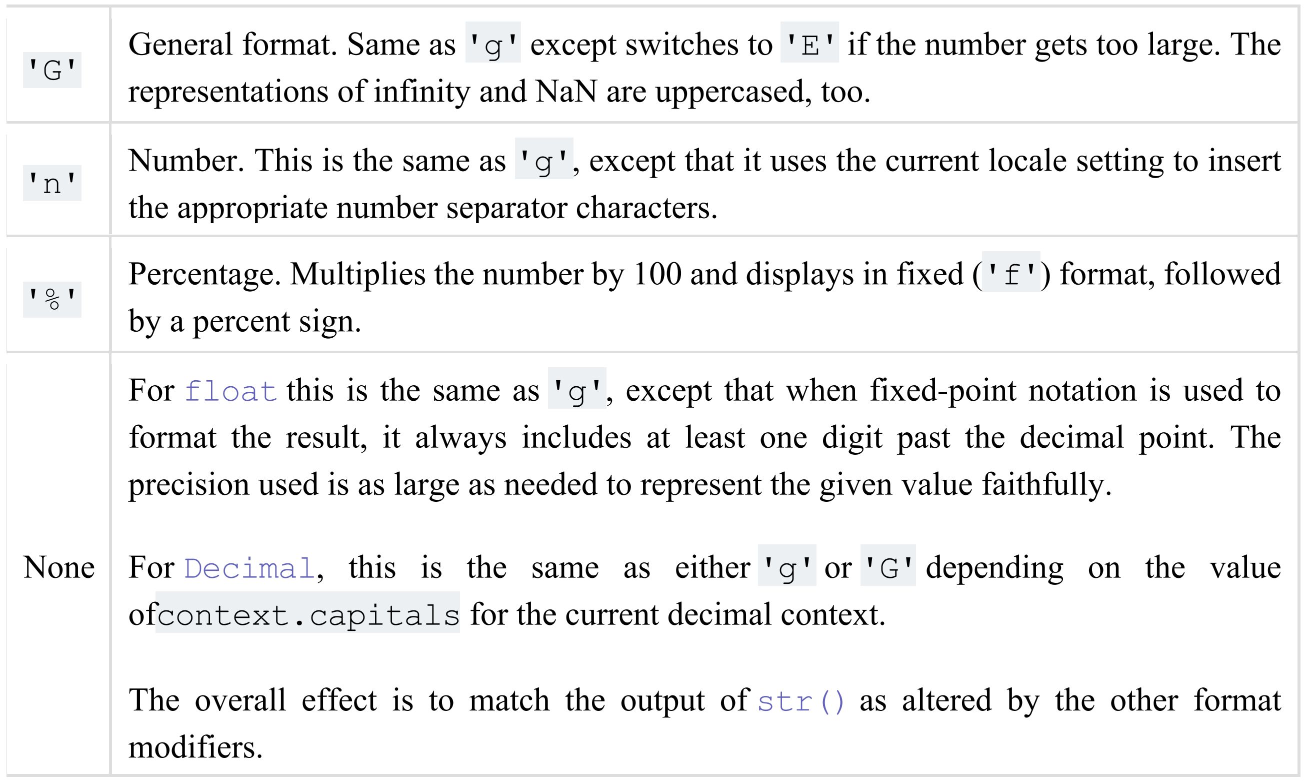
The available presentation types for *float* and *Decimal* values are:

Type	Meaning
`'e'`	Scientific notation. For a given precision `p`, formats the number in scientific notation with the letter 'e' separating the coefficient from the exponent. The coefficient has one digit before and `p` digits after the decimal point, for a total of `p + 1` significant digits. With no precision given, uses a precision of `6` digits after the decimal point for `float`, and shows all coefficient digits for `Decimal`. If no digits follow the decimal point, the decimal point is also removed unless the `#` option is used.
`'E'`	Scientific notation. Same as `'e'` except it uses an upper case 'E' as the separator character.
`'f'`	Fixed-point notation. For a given precision `p`, formats the number as a decimal number with exactly `p` digits following the decimal point. With no precision given, uses a precision of `6` digits after the decimal point for `float`, and uses a precision large enough to show all coefficient digits for `Decimal`. If no digits follow the decimal point, the decimal point is also removed unless the `#` option is used.
`'F'`	Fixed-point notation. Same as `'f'`, but converts `nan` to `NAN` and `inf` to `INF`.

Type	Meaning
`'g'`	General format. For a given precision `p >= 1`, this rounds the number to `p`significant digits and then formats the result in either fixed-point format or in scientific notation, depending on its magnitude. A precision of `0` is treated as equivalent to a precision of `1`. The precise rules are as follows: suppose that the result formatted with presentation type `'e'` and precision `p-1` would have exponent `exp`. Then, if `m <=exp < p`, where `m` is -4 for floats and -6 for `Decimals`, the number is formatted with presentation type `'f'` and precision `p-1-exp`. Otherwise, the number is formatted with presentation type `'e'` and precision `p-1`. In both cases insignificant trailing zeros are removed from the significand, and the decimal point is also removed if there are no remaining digits following it, unless the `'#'` option is used. With no precision given, uses a precision of `6` significant digits for `float`. For`Decimal`, the coefficient of the result is formed from the coefficient digits of the value; scientific notation is used for values smaller than `1e-6` in absolute value and values where the place value of the least significant digit is larger than 1, and fixed-point notation is used otherwise. Positive and negative infinity, positive and negative zero, and nans, are formatted as `inf`, `-inf`, `0`, `-0` and `nan` respectively, regardless of the precision.

7.1.1. Format Specification Mini-Language (cont.)

The available presentation types for *float* and *Decimal* values are:



`'G'`	General format. Same as `'g'` except switches to `'E'` if the number gets too large. The representations of infinity and NaN are uppercased, too.
`'n'`	Number. This is the same as `'g'`, except that it uses the current locale setting to insert the appropriate number separator characters.
`'%'`	Percentage. Multiplies the number by 100 and displays in fixed (`'f'`) format, followed by a percent sign.
None	For `float` this is the same as `'g'`, except that when fixed-point notation is used to format the result, it always includes at least one digit past the decimal point. The precision used is as large as needed to represent the given value faithfully. For `Decimal`, this is the same as either `'g'` or `'G'` depending on the value of`context.capitals` for the current decimal context. The overall effect is to match the output of `str()` as altered by the other format modifiers.

7.1.2. The String `format()` Method

- The brackets and characters within them (called format fields) are replaced with the objects passed into the `str.format()` method

```
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

- A number in the brackets can be used to refer to the position of the object passed into the `str.format()` method

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

- If keyword arguments are used in the `str.format()` method, their values are referred to by using the name of the argument

```
>>> print('This {food} is {adjective}.'.format(
...       food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.
```

- Positional and keyword arguments can be arbitrarily combined

```
>>> print('The story of {0}, {1}, and {other}.'.format('Bill',
'Manfred',

other='Georg'))
The story of Bill, Manfred, and Georg.
```

7.1.2. The String `format()` Method (cont.)

- If you have a real long format string that you don't want to split up, it would be nice if you could reference the variables to be formatted by name instead of by position
- This can be done by simply passing the dict and using square brackets '`[]`' to access the keys
- This could also be done by passing the table as keyword arguments with the '`**`' notation
- This is particularly useful in combination with the built-in function `vars()`, which returns a dictionary containing all local variables
- As an example, the following lines produce a tidily-aligned set of columns giving integers and their squares and cubes

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...       'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab:
{Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

```
>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

7.1.2. Format String Syntax

https://docs.python.org/release/3.9.6/library/string.html#formatstrings

```
replacement_field ::=  "{" [field_name] ["!" conversion] [":" format_spec] "}"
field_name        ::=  arg_name ("." attribute_name | "[" element_index "]")*
arg_name          ::=  [identifier | digit+]
attribute_name    ::=  identifier
element_index     ::=  digit+ | index_string
index_string      ::=  <any source character except "]"> +
conversion        ::=  "r" | "s" | "a"
format_spec       ::=  <described in the next section>
```

7.1.3. Manual String Formatting

- One space between each column was added by the way `print()` works: it always adds spaces between its arguments
- The `str.rjust()` method of string objects right-justifies a string in a field of a given width by padding it with spaces on the left
- There are similar methods `str.ljust()` and `str.center()`
- These methods do not write anything, they just return a new string
- If the input string is too long, they don't truncate it, but return it unchanged
- This will mess up your column lay-out but that's usually better than the alternative, which would be lying about a value
- If you really want truncation you can always add a slice operation, as in `x.ljust(n)[:n]`
- There is another method, `str.zfill()`, which pads a numeric string on the left with zeros (it understands about plus and minus signs)

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Note use of 'end' on previous line
...     print(repr(x*x*x).rjust(4))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

7.1.4. Old `string` Formatting

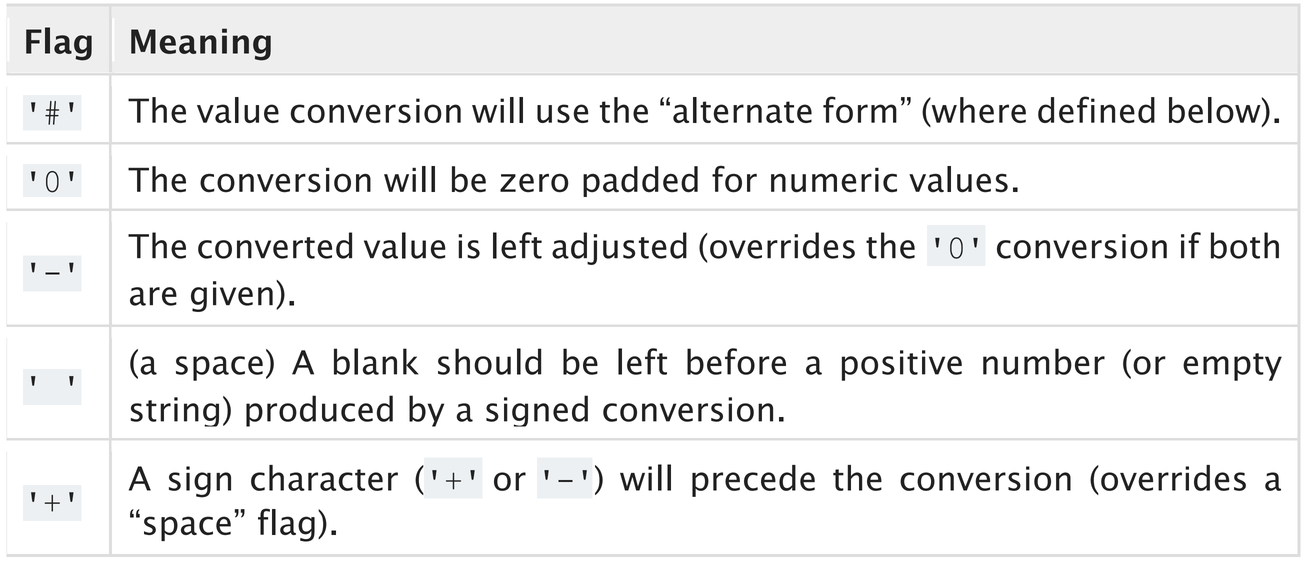
- The `%` operator (modulo) can also be used for string formatting
- Given `'string' % values`, instances of `%` in `string` are replaced with zero or more elements of `values`
- This operation is commonly known as string interpolation
- When the right argument is a dictionary (or other mapping type), then the formats in the string must include a parenthesized mapping key into that dictionary inserted immediately after the '`%`' character
- The mapping key selects the value to be formatted from the mapping

```
>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
The value of pi is approximately 3.142.
```

```
>>> print('%(language)s has %(number)03d quote types.' %
...       {'language': "Python", "number": 2})
Python has 002 quote types.
```

7.1.2. `printf`-style String Formatting

https://docs.python.org/release/3.9.6/library/stdtypes.html#old-string-formatting



Flag	Meaning
`'#'`	The value conversion will use the "alternate form" (where defined below).
`'0'`	The conversion will be zero padded for numeric values.
`'-'`	The converted value is left adjusted (overrides the `'0'` conversion if both are given).
`' '`	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
`'+'`	A sign character (`'+'` or `'-'`) will precede the conversion (overrides a "space" flag).

7.1.2. `printf`-style String Formatting (cont.)

Conversion	Meaning
`'d'`	Signed integer decimal.
`'i'`	Signed integer decimal.
`'o'`	Signed octal value.
`'u'`	Obsolete type – it is identical to `'d'`.
`'x'`	Signed hexadecimal (lowercase).
`'X'`	Signed hexadecimal (uppercase).
`'e'`	Floating point exponential format (lowercase).
`'E'`	Floating point exponential format (uppercase).
`'f'`	Floating point decimal format.
`'F'`	Floating point decimal format.
`'g'`	Floating point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.
`'G'`	Floating point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.
`'c'`	Single character (accepts integer or single character string).
`'r'`	String (converts any Python object using `repr()`).
`'s'`	String (converts any Python object using `str()`).
`'a'`	String (converts any Python object using `ascii()`).
`'%'`	No argument is converted, results in a `'%'` character in the result.

7.2. Reading and Writing Files

- `open()` returns a *file object*, and is most commonly used with two arguments: `open(filename, mode)`
- The first argument is a string containing the filename
- The second argument is another string containing a few characters describing the way in which the file will be used
- '`r`' opens the file for read only
- '`w`' opens the file for writing only - an existing file with the same name will be erased
- '`a`' opens the file for appending - any data written to the file is automatically added to the end
- '`r+`' opens the file for both reading and writing
- The mode argument is optional and '`r`' will be assumed if it's omitted

```
>>> f = open('workfile', 'w')
```

- Normally, files are opened in *text mode*
- When reading in text mode, the default is to convert platform-specific line endings (`\n` on Unix, `\r\n` on Windows) to just `\n`
- When writing in text mode, the default is to convert occurrences of `\n` back to platform-specific line endings
- '`b`' appended to the mode opens the file in *binary mode*: now the data is read and written in the form of bytes objects
- This mode should be used for all files that don't contain text.
- Note: The text mode line ending conversions mentioned above will corrupt binary data like that in JPEG or EXE files

7.2. Reading and Writing Files (cont.)

- It is good practice to use the `with` keyword when dealing with file objects
- The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point
- If you're not using the `with` keyword, then you should call `f.close()` to close the file and immediately free up any system resources used by it
- After a file object is closed, either by a `with` statement or by calling `f.close()`, attempts to use the file object will automatically fail

```
>>> with open('workfile') as f:
...     read_data = f.read()

>>> # We can check that the file has been automatically closed.
>>> f.closed
True
```

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

7.2.1. Methods of File Objects

- To read a file's contents, call `f.read(size)`, which reads some quantity of data and returns it as a string (in *text mode*) or bytes object (in *binary mode*)
- `size` is an optional numeric argument
- When `size` is omitted or negative, the entire contents of the file will be read and returned; it's your problem if the file is twice as large as your machine's memory
- Otherwise, at most *size* characters (in text mode) or size bytes (in binary mode) are read and returned
- If the end of the file has been reached, `f.read()` will return an empty string ('')

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

7.2.1. Methods of File Objects (cont.)

- `f.readline()` reads a single line from the file; a newline character (`\n`) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline
- if `f.readline()` returns an empty string, the end of the file has been reached, while a blank line is represented by `'\n'`, a string containing only a single newline
- For reading lines from a file, you can loop over the file object
- If you want to read all the lines of a file in a list you can also use `list(f)` or `f.readlines()`
- `f.write(string)` writes the contents of *string* to the file, returning the number of characters written
- Other types of objects need to be converted – either to a string (in text mode) or a bytes object (in binary mode) – before writing them

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

```
>>> f.write('This is a test\n')
15
```

```
>>> value = ('the answer', 42)
>>> s = str(value)  # convert the tuple to string
>>> f.write(s)
18
```

7.2.1. Methods of File Objects (cont.)

- `f.tell()` returns an integer giving the file object's current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode
- To change the file object's position, use `f.seek(offset, whence)`
- The position is computed from adding offset to a reference point; the reference point is selected by the *whence* argument
- A *whence* value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point
- *Whence* can be omitted and defaults to 0, using the beginning of the file as the reference point

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

7.2.2. Saving structured data with `json`

- Python allows you to use the popular data interchange format called *JSON (JavaScript Object Notation)*
- The standard module called `json` can take Python data hierarchies, and convert them to string representations; this process is called serializing
- Reconstructing the data from the string representation is called deserializing
- Between serializing and deserializing, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine
- If you have an object `x`, you can view its JSON string representation with a simple line of code
- Another variant of the `dumps()` function, called `dump()`, simply serializes the object to a text file
- If `f` is a *text file* object opened for writing, we can serialize `x` to a *JSON*
- If `f` is a *text file* object opened for reading, we can deserialize the JSON into `x`
- This simple serialization technique can handle lists and dictionaries, but serializing arbitrary class instances in JSON requires a bit of extra effort

```
>>> import json
>>> x = [1, 'simple', 'list']
>>> json.dumps(x)
'[1, "simple", "list"]'
```

```
json.dump(x, f)
```

```
x = json.load(f)
```