Mantas Sinkevičius - Unsplash

Strings and string manipulation


embarrassing
omission
An new foe has appeared!
#
CHALLENGER APPROACHING

Comments

Anything after a # in a line of your code is ignored

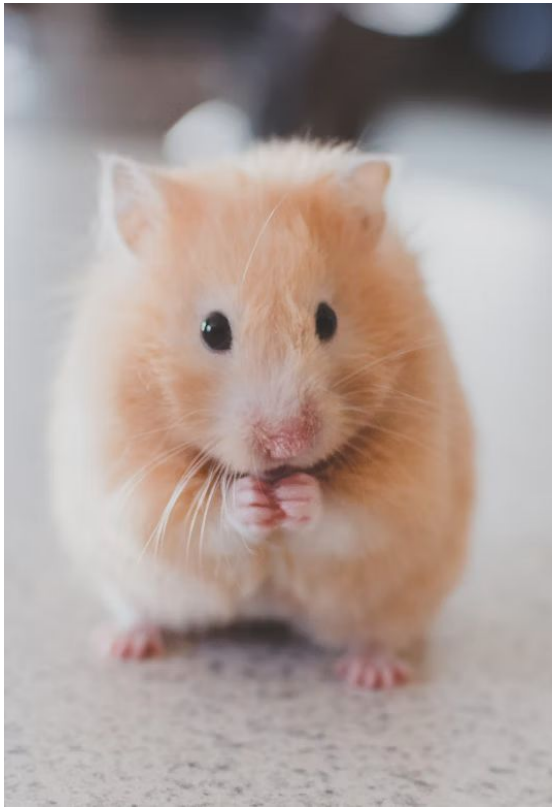
Comments

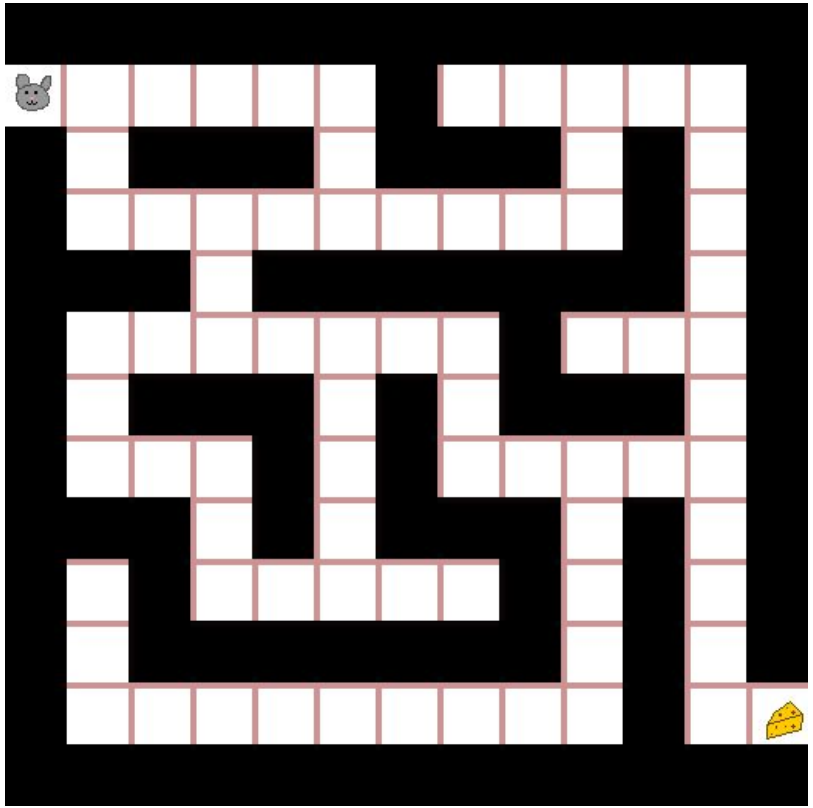
Anything after a # in a line of your code is ignored

```
x= 7 # this is ignored
# this is ignored too!
# x = 25
print(x)
print(“These symbols in strings # don’t count”)
```

Activity!

Code on Blackboard!





Refresher

What is a string?

One of our data types, a sequence of characters (letters, #s, symbols)

Strings must begin and end with a matching pair of “ or ‘

```
university_name = “gvsu”
```

Indexing

Strings are a *sequence* type

We can do cool things with sequences!

Indexing

```
example_str = “this is a test!”
```

Indexing

```
example_str = “this is a test!”
```

We can *index* into a string to access one or more characters!

Indexing

```
example_str = “this is a test!”
```

We can *index* into a string to access one or more characters!

Indexing starts at 0

Indexing

```
example_str = “this is a test!”
```

We can *index* into a string to access one or more characters!

Indexing starts at 0

```
example_str[0]
example_str[1]
example_str[6]
```

Indexing

```
example_str = “this is a test!”
```

We can *index* into a string to access one or more characters!

Indexing starts at 0

```
example_str[0] -> “t”
example_str[1]
example_str[6]
```

Indexing

```
example_str = “this is a test!”
```

We can *index* into a string to access one or more characters!

Indexing starts at 0

```
example_str[0] -> “t”
example_str[1] -> “h”
example_str[6]
```

Indexing

```
example_str = “this is a test!”
```

We can *index* into a string to access one or more characters!

Indexing starts at 0

```
example_str[0] -> “t”
example_str[1] -> “h”
example_str[6] -> “s”
```

Indexing

```
example_str = “this is a test!”


example_str[0] -> “t”
example_str[1] -> “h”
example_str[6] -> “s”
len(example_str)
example_str[15]
example_str[14]
example_str[-1]
example_str[-2]
```

Indexing

```
example_str = “this is a test!”


example_str[0] -> “t”
example_str[1] -> “h”
example_str[6] -> “s”
len(example_str) -> 15 (number of characters in string)
example_str[15]
example_str[14]
example_str[-1]
example_str[-2]
```

Indexing

```
example_str = “this is a test!”


example_str[0] -> “t”
example_str[1] -> “h”
example_str[6] -> “s”
len(example_str) -> 15
example_str[15] -> ERROR! Index out of bounds!
example_str[14]
example_str[-1]
example_str[-2]
```

Indexing

```
example_str = “this is a test!”


example_str[0] -> “t”
example_str[1] -> “h”
example_str[6] -> “s”
len(example_str) -> 15
example_str[15] -> ERROR! Index out of bounds!
example_str[14] -> “!”
example_str[-1]
example_str[-2]
```

Indexing

```
example_str = “this is a test!”


example_str[0] -> “t”
example_str[1] -> “h”
example_str[6] -> “s”
len(example_str) -> 15
example_str[15] -> ERROR! Index out of bounds!
example_str[14] -> “!”
example_str[-1] -> “!”
example_str[-2]
```

Indexing

```
example_str = “this is a test!”


example_str[0] -> “t”
example_str[1] -> “h”
example_str[6] -> “s”
len(example_str) -> 15
example_str[15] -> ERROR! Index out of bounds!
example_str[14] -> “!”
example_str[-1] -> “!”
example_str[-2] -> “t”
```

Assigning via indexing

```
pet = “cat”
pet[0] = “b”
```

What is the value of `pet`?

Assigning via indexing

```
pet = “cat”
pet[0] = “b”
```

What is the value of `pet`?

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

Assigning via indexing

```
pet = “cat”
pet[0] = “b”
```

What is the value of `pet`?

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

It’s a trick! Strings are **immutable**, they cannot be changed!

Advanced indexing -> Slicing

We can pull out more than one character!

Advanced indexing -> Slicing

We can pull out more than one character!

```
string_var[a:b]
```

Advanced indexing -> Slicing

We can pull out more than one character!

```
string_var[a:b]
```

Start at index a (included)

Advanced indexing -> Slicing

We can pull out more than one character!

```
string_var[a:b]
```

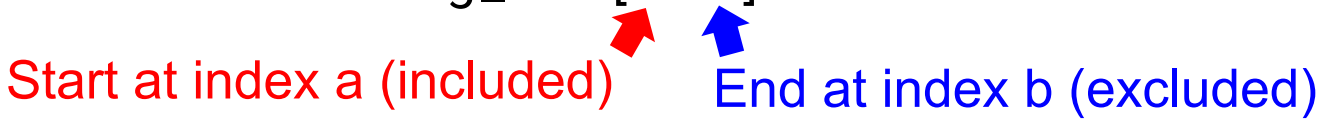
Start at index a (included)

End at index b (excluded)

Advanced indexing -> Slicing

We can pull out more than one character!

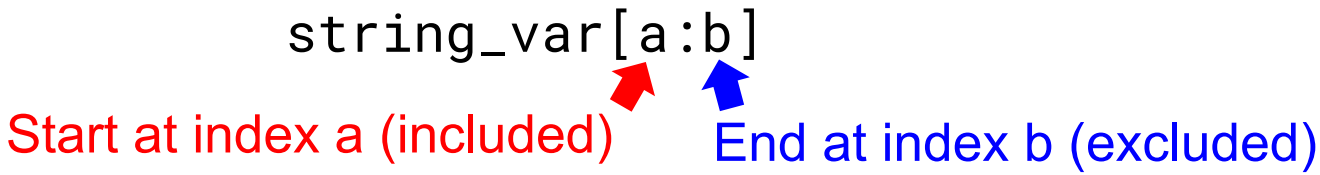
```
string_var[a:b]
```




```
example_str = “this is a test!”
example_str[1:3]
```

Advanced indexing -> Slicing

We can pull out more than one character!

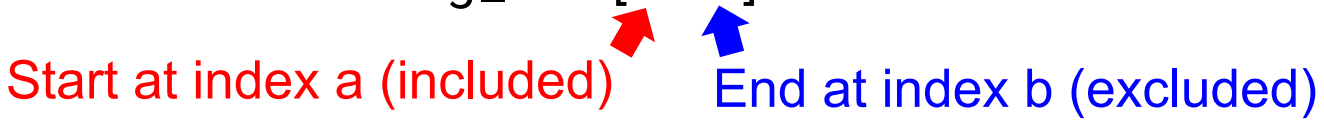



```
example_str = “this is a test!”
example_str[1:3] -> “hi”
```

Advanced indexing -> Slicing

We can pull out more than one character!

```
string_var[a:b]
```



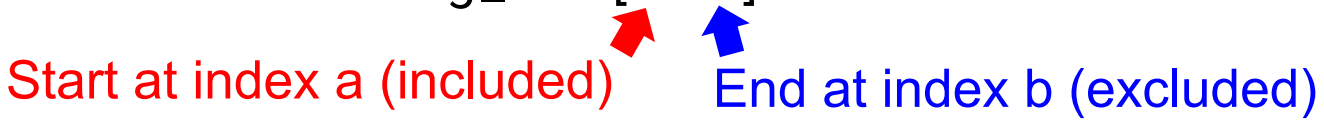

```
example_str = “this is a test!”
example_str[1:3] -> “hi”
```

What would the code be to extract the word “test”?

Advanced indexing -> Slicing

We can pull out more than one character!

```
string_var[a:b]
```




```
example_str = “this is a test!”
example_str[1:3] -> “hi”
```

What would the code be to extract the word “test”?

```
example_str[10:14]
```

Even more slicing!

We can leave out indices:

Even more slicing!

We can leave out indices:

`string_var[a:]` -> start at index a (included), go to end

Even more slicing!

We can leave out indices:

`string_var[a:]` -> start at index a (included), go to end

`string_var[:b]` -> start at beginning, go to b (excluded)

Even more slicing!

We can leave out indices:

`string_var[a:]` -> start at index a (included), go to end

`string_var[:b]` -> start at beginning, go to b (excluded)

`string_var[:]` -> whole string!

Even more slicing!

We can leave out indices:

`string_var[a:]` -> start at index a (included), go to end

`string_var[:b]` -> start at beginning, go to b (excluded)

`string_var[:]` -> whole string!

`string_var[a:b:c]` -> slice from a to b, stepping by c (stride)

Even more slicing!

We can leave out indices:

`string_var[a:]` -> start at index a (included), go to end

`string_var[:b]` -> start at beginning, go to b (excluded)

`string_var[:]` -> whole string!

`string_var[a:b:c]` -> slice from a to b, stepping by c (stride)

You can also use negative indices in slices

Strings and operators

What do you think this does?

```
“A” + “part”
```

Strings and operators

What do you think this does?

`“A” + “part” -> “Apart”`

Adding two strings *concatenates* them!

Strings and operators

What are the values of these strings at the end of execution?

```
str_1 = “the”
str_2 = “end”
str_1 = str_1 + “ ” + str_2
```

Strings and operators

What are the values of these three strings at the end of execution?

```
str_1 = “the”
str_2 = “end”
str_1 = str_1 + “ ” + str_2
```

`str_1 -> “the end”; str_2 -> “end”` (unchanged)

Strings and operators

What are the values of these three strings at the end of execution?

```
str_1 = “the”
str_2 = “end”
str_1 = str_1 + “ ” + str_2
```

`str_1 -> “the end”; str_2 -> “end”` (unchanged)

Wait! We said we can’t change strings???

Strings and operators

What are the values of these three strings at the end of execution?

```
str_1 = “the”
str_2 = “end”
str_1 = str_1 + “ ” + str_2
```

`str_1 -> “the end”; str_2 -> “end”` (unchanged)

Wait! We said we can’t change strings???

Concatenation creates a copy of the string (which is changed)

And then we store the copy back into str_1

Strings and other operators

What do you think this does?

```
“AAA” - “A”
```

Strings and other operators

What do you think this does?

“AAA” - “A”

```
>>> "AAA" - "A"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Addition works on strings, but subtraction doesn’t!

Strings and other operators

What do you think this does?

```
“AAA” + 12
```

Strings and other operators

What do you think this does?

“AAA” + 12

```
>>> "AAA" + 12
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

You can add (concatenate) two strings, but not string + int or string + float!

Strings and other operators

What do you think this does?

```
“A” * 3
```

Strings and other operators

What do you think this does?

`“A” * 3 -> “AAA”`

Concatenates N copies of the string together

Strings and other operators

What do you think this does?

`“A” * 3 -> “AAA”`

Concatenates N copies of the string together

`4 * 3`

Strings and other operators

What do you think this does?

`“A” * 3 -> “AAA”`

Concatenates N copies of the string together

4 * 3 -> 4 + 4 + 4

Strings and other operators

What do you think this does?

`“A” * 3 -> “AAA”`

Concatenates N copies of the string together

4 * 3 -> 4 + 4 + 4

“A” * 3

Strings and other operators

What do you think this does?

`“A” * 3 -> “AAA”`

Concatenates N copies of the string together

4 * 3 -> 4 + 4 + 4

“A” * 3 -> “A” + “A” + “A”

Strings and other operators

These also don’t work:

“A” * “A”

“A” / “A”

“A” / 2

“A” - 3

String methods

Methods are *functions* associated with an *object*

String methods

Methods are *functions* associated with an *object*

What will this code output?

```
example_str = “this is a test!”
count = example_str.count(“s”)
print(count)
```

String methods

Methods are *functions* associated with an *object*

What will this code output?

```
example_str = “this is a test!”
count = example_str.count(“s”)
print(count)
```

Should print 3, as there are three instances of “s”

String methods

Methods are *functions* associated with an *object*

What will this code output?

```
example_str = “this is a test!”
count = example_str.count(“s”)
print(count)
```

Should print 3, as there are three instances of “s”
What about this one?

```
print(example_str.count(“is”))
```

String methods

Methods are *functions* associated with an *object*

What will this code output?

```
example_str = “this is a test!”
count = example_str.count(“s”)
print(count)
```

Should print 3, as there are three instances of “s”
What about this one?
`print(example_str.count(“is”)) -> 2 (this is)`

String methods

```
claim = “you will pass the exam if you study your notes”
claim.replace(“you”, “YOU”)
print(claim)
```

String methods

```
claim = “you will pass the exam if you study your notes”
claim.replace(“you”, “YOU”)
print(claim)
```

Remember: methods cannot modify strings in place!

String methods

```
claim = “you will pass the exam if you study your notes”
claim.replace(“you”, “YOU”)
print(claim)
```

Remember: methods cannot modify strings in place!

In this example, claim does not change!

String methods

```
claim = “you will pass the exam if you study your notes”
claim.replace(“you”, “YOU”)
print(claim)
```

Remember: methods cannot modify strings in place!
In this example, claim does not change!
Working version:

```
claim = “you will pass the exam if you study your notes”
new_claim = claim.replace(“you”, “YOU”)
print(new_claim)
```

String methods

```
example_str = “Hello, how are YOU?”
example_str.upper()
example_str.lower()
example_str.capitalize()
```

String methods

```
example_str = “Hello, how are YOU?”
example_str.upper() -> “HELLO, HOW ARE YOU?”
example_str.lower()
example_str.capitalize()
```

String methods

```
example_str = “Hello, how are YOU?”
example_str.upper() -> “HELLO, HOW ARE YOU?”
example_str.lower() -> “hello, how are you?”
example_str.capitalize()
```

String methods

```
example_str = “Hello, how are YOU?”
example_str.upper() -> “HELLO, HOW ARE YOU?”
example_str.lower() -> “hello, how are you?”
example_str.capitalize() -> “Hello, how are you?”
```

String methods

```
example_str = “    lots of whitespace                “
example_str.strip()
example_str.rstrip()
example_str.lstrip()
```

String methods

```
example_str = “    lots of whitespace                “
example_str.strip() -> “lots of whitespace”
example_str.rstrip()
example_str.lstrip()
```

String methods

```
example_str = “    lots of whitespace                “
example_str.strip() -> “lots of whitespace”
example_str.rstrip() -> “    lots of whitespace”
example_str.lstrip()
```

String methods

```
example_str = “    lots of whitespace                “
example_str.strip() -> “lots of whitespace”
example_str.rstrip() -> “    lots of whitespace”
example_str.lstrip() -> “lots of whitespace                “
```

String methods

```
example_str = “this is a test!”
example_str.find(“is”)
```

What do we expect this method to return?

String methods

```
example_str = “this is a test!”
example_str.find(“is”)
```

What do we expect this method to return?

It returns 2, the starting index of the first instance of “is”

String methods

```
example_str = “this is a test!”
example_str.find(“is”)
```

What do we expect this method to return?

It returns 2, the starting index of the first instance of “is”

There are also *optional* parameters:

```
string_var.find(pattern, start, end)
```

String methods

```
example_str = “this is a test!”
example_str.find(“is”) -> 2
example_str.find(“is”, 3)
example_str.find(“is”, 6)
```

String methods

```
example_str = “this is a test!”
example_str.find(“is”) -> 2
example_str.find(“is”, 3)
example_str.find(“is”, 6)
```

String methods

```
example_str = “this is a test!”
example_str.find(“is”) -> 2
example_str.find(“is”, 3) -> 5
example_str.find(“is”, 6)
```

String methods

```
example_str = “this is a test!”
example_str.find(“is”) -> 2
example_str.find(“is”, 3) -> 5
example_str.find(“is”, 6) -> -1 (no match)
```

String methods

There are MANY string methods

Do you know where you can access them all?

String methods

There are MANY string methods

Do you know where you can access them all?

In the docs :^)
https://docs.python.org/3/library/stdtypes.html#string-methods

More on F-Strings

What is an F-String?

More on F-Strings

What is an F-String?

A *formatted string literal*

Really just a string that have evaluated expressions injected into it!

More on F-Strings

What is an F-String?

A *formatted string literal*

Really just a string that have evaluated expressions injected into it!

What have they looked like so far?

More on F-Strings

What is an F-String?

A *formatted string literal*

Really just a string that have evaluated expressions injected into it!

What have they looked like so far?

```
x = 7
print(f“The value of x is {x}”)
```

More on F-Strings

F-strings do not need to be in a call to print!

More on F-Strings

F-strings do not need to be in a call to print!

```
my_str = f”this is a string! The value of x is {x}”
```

More on F-Strings

So far, you've likely only had one expression within the F-string

You can have more!

What is the value of the string here?

```
x = 3
y = 4
equation = f"{x} * {y} = {x * y}"
```

All about quotes

Strings must start and end with a matching pair of either “ or ‘

What if you want to include quotes within a string?

All about quotes

Strings must start and end with a matching pair of either “ or ‘

What if you want to include quotes within a string?

Easiest way: use the *other* quotes on the outside of the string.

All about quotes

Strings must start and end with a matching pair of either “ or ‘

What if you want to include quotes within a string?

Easiest way: use the *other* quotes on the outside of the string.

```
str_1 = “I can use ‘single quotes’ in here”
```

All about quotes

Strings must start and end with a matching pair of either “ or ‘

What if you want to include quotes within a string?

Easiest way: use the *other* quotes on the outside of the string.

```
str_1 = “I can use ‘single quotes’ in here”
str_2 = ‘And “vice-versa”!’
```

Special characters

There are some weird characters that you want to put in a string.

Most common example: a newline

Special characters

There are some weird characters that you want to put in a string.

Most common example: a newline

How do we say “I want a newline here”?

Special characters - Escape sequences

There are some weird characters that you want to put in a string.

Most common example: a newline

How do we say “I want a newline here”? Escape sequences!

Special characters - Escape sequences

Running this code:

```
print(“This is line one\nThis is line two”)
```

Special characters - Escape sequences

Running this code:

```
print(“This is line one\nThis is line two”)
```

Will result in:

```
This is line one
This is line two
```

Special characters - Escape sequences

Running this code:

```
print(“This is line one\nThis is line two”)
```

Will result in:

```
This is line one
This is line two
```

In python, \n (backslash n) is how we specify a newline!

Special characters - Escape sequences

Common escape sequences: (docs)

`\n` - newline
`\t` - tab
`\”` and `\’` - Escaped quotes, do NOT end a string

What if I want an actual backslash in my string?

Special characters - Escape sequences

Common escape sequences: (docs)

`\n` - newline
`\t` - tab
`\”` and `\’` - Escaped quotes, do NOT end a string

What if I want an actual backslash in my string?

`\\` - First one starts an escape sequence, second turns it into a single backslash

Formatting strings: format specifiers

You can precisely specify the format of your injected values

Formatting strings: format specifiers

You can precisely specify the format of your injected values

Examples:

```
f”x = {2:b}”
```

Formatting strings: format specifiers

You can precisely specify the format of your injected values

Examples:

`f”x = {2:b}”` -> Print as binary (10)

Formatting strings: format specifiers

You can precisely specify the format of your injected values

Examples:

`f”x = {2:b}”` -> Print as binary (10)

`f”x = {2:08}”` (or 08d)

Formatting strings: format specifiers

You can precisely specify the format of your injected values

Examples:

`f”x = {2:b}”` -> Print as binary (10)

`f”x = {2:08}”` (or 08d) -> Zero pad to a total of 8 digits (00000002)

Formatting strings: format specifiers

You can precisely specify the format of your injected values

Examples:

`f”x = {2:b}”` -> Print as binary (10)

`f”x = {2:08}”` (or 08d) -> Zero pad to a total of 8 digits (00000002)

`f”x = {112:e}”`

Formatting strings: format specifiers

You can precisely specify the format of your injected values

Examples:

`f"x = {2:b}"` -> Print as binary (10)

`f"x = {2:08}"` (or 08d) -> Zero pad to a total of 8 digits (00000002)

`f"x = {112:e}"` -> Scientific notation (1.120000e+2)

Formatting strings: format specifiers

You can precisely specify the format of your injected values

Examples:

`f”x = {2:b}”` -> Print as binary (10)

`f”x = {2:08}”` (or 08d) -> Zero pad to a total of 8 digits (00000002)

`f”x = {112:e}”` -> Scientific notation (1.120000e+2)

`f”x = {112:g}”`

Formatting strings: format specifiers

You can precisely specify the format of your injected values

Examples:

`f"x = {2:b}"` -> Print as binary (10)

`f"x = {2:08}"` (or 08d) -> Zero pad to a total of 8 digits (00000002)

`f"x = {112:e}"` -> Scientific notation (1.120000e+2)

`f"x = {112:g}"` -> Choose between normal and scientific notation

Formatting strings: format specifiers

You can precisely specify the format of your injected values

Examples:

`f”x = {2:b}”` -> Print as binary (10)

`f”x = {2:08}”` (or 08d) -> Zero pad to a total of 8 digits (00000002)

`f”x = {112:e}”` -> Scientific notation (1.120000e+2)

`f”x = {112:g}”` -> Choose between normal and scientific notation

```
f”pi = {math.pi:.03f}”
```

Formatting strings: format specifiers

You can precisely specify the format of your injected values

Examples:

`f"x = {2:b}"` -> Print as binary (10)

`f"x = {2:08}"` (or 08d) -> Zero pad to a total of 8 digits (00000002)

`f"x = {112:e}"` -> Scientific notation (1.120000e+2)

`f"x = {112:g}"` -> Choose between normal and scientific notation

`f"pi = {math.pi:.03f}"` -> Round to three decimal places (3.142)

Formatting strings: format specifiers

You can precisely specify the format of your injected values

Examples:

`f"x = {2:b}"` -> Print as binary (10)

`f"x = {2:08}"` (or 08d) -> Zero pad to a total of 8 digits (00000002)

`f"x = {112:e}"` -> Scientific notation (1.120000e+2)

`f"x = {112:g}"` -> Choose between normal and scientific notation

`f"pi = {math.pi:.03f}"` -> Round to three decimal places (3.142)

Full docs: here