

Python notes part 2

```
17 / 3 # classic division returns a float

17 // 3 # floor division discards the fractional part

17 % 3 # the % operator returns the remainder of the division

5 * 3 + 2 # floored quotient * divisor + remainder
```

```
5 ** 2 # 5 squared

2 ** 7 # 2 to the power of 7
```

The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
width = 20
height = 5 * 9
width * height
```

In interactive mode, the last printed expression is assigned to the variable `_`. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```
tax = 12.5 / 100
price = 100.50
price * tax

price + _

round(_, 2)
```

To quote a quote, we need to “escape” it, by preceding it with `\`. Alternatively, we can use the other type of quotation marks:

```
>>> 'doesn\'t' # use \' to escape the single quote...
"doesn't"
>>> "doesn't" # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

In the Python shell, the string definition and output string can look different. The `print()` function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
>>> s = 'First line.\nSecond line.' # \n means newline
>>> s # without print(), special characters are included in the string
'First line.\nSecond line.'
>>> print(s) # with print(), special characters are interpreted, so \n produces new line
First line.
Second line.
```

If you don’t want characters prefaced by `\` to be interpreted as special characters, you can use *raw strings* by adding an `r` before the first quote:

```
>>> print('C:\some\name') # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name') # note the r before the quote
C:\some\name
```

There is one subtle aspect to raw strings: a raw string may not end in an odd number of `\` characters; see [the FAQ entry](#) for more information and workarounds.

String literals can span multiple lines. One way is using triple-quotes: `"""..."""` or `'''...'''`. End of lines are automatically included in the string, but it’s possible to prevent this by adding a `\` at the end of the line. The following example:

```
print("""\
Usage: thingy [OPTIONS]
    -h            Display this usage message
    -H hostname  Hostname to connect to
""")
```

Strings can be concatenated (glued together) with the `+` operator, and repeated with `*`:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Two or more *string literals* (i.e. the ones enclosed between quotes) next to each other are automatically concatenated.

```
>>> 'Py' 'thon'
'Python'
```

If you want to concatenate variables or a variable and a literal, use `+`:

```
>>> prefix + 'thon'
'Python'
```

Indices may also be negative numbers, to start counting from the right:

```
>>> word[-1] # last character
'n'
>>> word[-2] # second-last character
'o'
>>> word[-6]
'p'
```

In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain a substring:

```
>>> word[0:2] # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5] # characters from position 2 (included) to 5 (excluded)
'tho'
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
>>> word[:2] # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:] # characters from position 4 (included) to the end
'on'
>>> word[-2:] # characters from the second-last (included) to the end
'on'
```

Python strings cannot be changed — they are [immutable](#). Therefore, assigning to an indexed position in the string results in an error:

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

If you need a different string, you should create a new one:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

The built-in function `len()` returns the length of a string:

The built-in function `len()` returns the length of a string:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Step is between the numbers

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the 'step'):

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

The status is a number like 400 or 404 and match MATCHES them

```
def http_error(status):  
    match status:  
        case 400:  
            return "Bad request"  
        case 404:  
            return "Not found"  
        case 418:  
            return "I'm a teapot"  
        case _:  
            return "Something's wrong with the internet"
```

Note the last block: the "variable name" `_` acts as a *wildcard* and never fails to match. If no case matches, none of the branches is executed.

This is text in document