

Python basics

- Comments
- `"""`
- Variable names
- `int`, `float`, `str`
- type conversion
- assignment (`=`)
- `print()`, `help()`, `type()`

Python comments

A '#' indicates the beginning of a comment. From '#' until the end of line is ignored by Python.

```
x = 42    # and here goes the comment
```

Comments useful to describe what a piece of code is supposed to do, what kind of input is expected, what is the output, side effects...

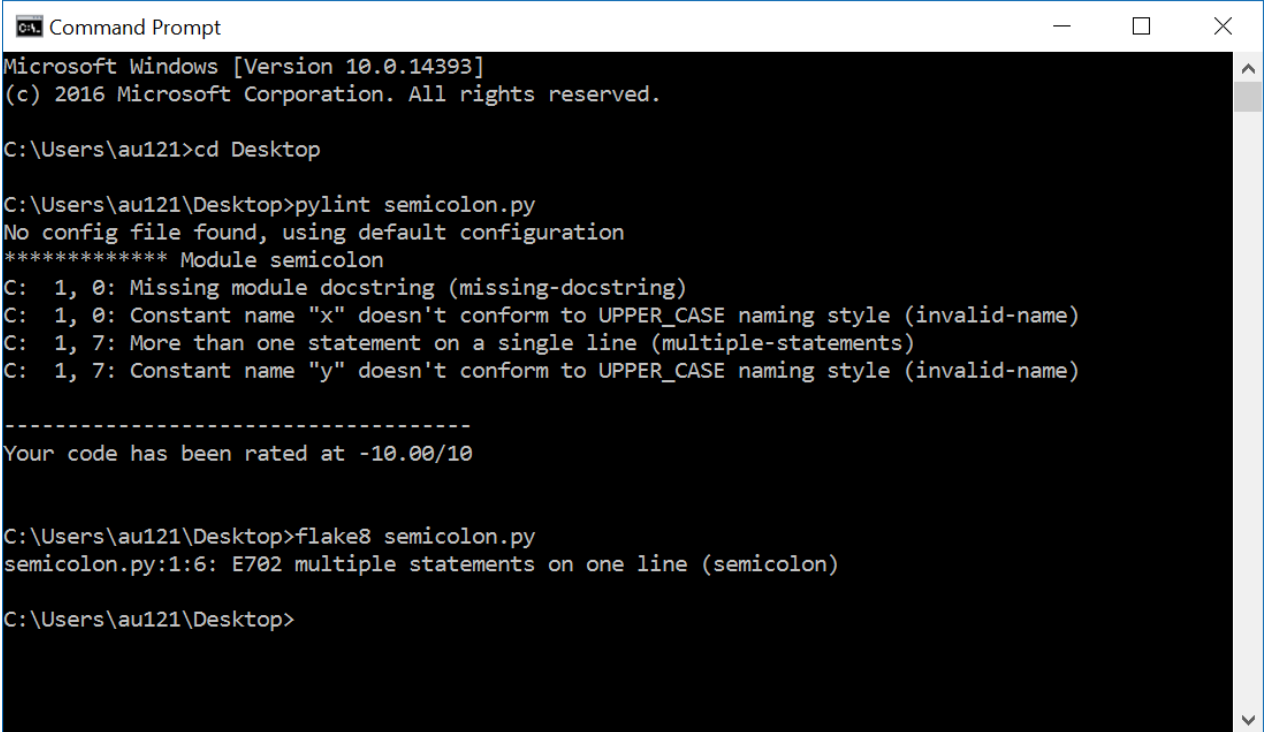
The “;” in Python

- Normally statements follow in consecutive lines with identical indentation

```
x = 1
y = 1
```

- but Python also allows multiple statements on one line, separated by “;”

```
x = 1; y = 1
```



```
Command Prompt
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\au121>cd Desktop

C:\Users\au121\Desktop>pylint semicolon.py
No config file found, using default configuration
***** Module semicolon
C:  1, 0: Missing module docstring (missing-docstring)
C:  1, 0: Constant name "x" doesn't conform to UPPER_CASE naming style (invalid-name)
C:  1, 7: More than one statement on a single line (multiple-statements)
C:  1, 7: Constant name "y" doesn't conform to UPPER_CASE naming style (invalid-name)

-----
Your code has been rated at -10.00/10

C:\Users\au121\Desktop>flake8 semicolon.py
semicolon.py:1:6: E702 multiple statements on one line (semicolon)

C:\Users\au121\Desktop>
```

neither **pylint** or **flake8** like “;”

- General Python PEP 8 guideline: **avoid using “;”**
- Other languages like C, C++ and Java require “;” to end/separate statements

Variable names

- Variable name = sequence of **letters** 'a'-'z', 'A'-'Z', **digits** '0'-'9', and **underscore** '_'


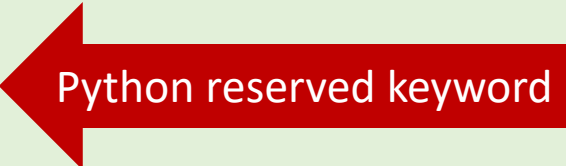
v, volume, height_of_box, WidthOfBox, x0, _v12_34B, _

(snake_case) (CamelCase)

- a name cannot start with a digit
- names are case sensitive (AB, Ab, aB and ab are different variables)


- Variable names are **references to objects in memory**
- **Use meaningful variables names**
- **Python 3 reserved keywords:** and, as, assert, break, class, continue, def, del, elif, else, except, False, finally, for, from, global, if, import, in, is, lambda, nonlocal, None, not, or, pass, raise, return, True, try, while, with, yield

Question – Not a valid Python variable name?

- a) `print`
-  b) `for`  Python reserved keyword
- c) `_100`
- d) `x`
- e) `_`
- f) `python_for_ever`
- g) Don't know

Python shell

```
> print = 7
> print(42)
| Traceback (most recent call last):
|   File "<stdin>", line 1, in <module>
|   TypeError: 'int' object is not callable
```



`print` is a valid variable name, with default value a builtin *function* to print output to a shell – assigning a new value to `print` is very likely a bad idea (like many others `sum`, `int`, `str`, ...)

Integer literals

- -4, -3, -2, -1, 0, 1, 2, 3, 4
- Python integers can have an arbitrary number of digits (only limited by machine memory)
- Can be preceded by a plus (+) or minus (-)
- For readability underscores (_) can be added between digits,

2_147_483_647

(for more, see [PEP 515 - Underscores in Numeric Literals](#))

Question – What statement will not fail?

a) $x = _42$




b) $_10 = -1_1$

c) $x = 1___0$

d) $x = +1_0_$

e) Don't know

Float literals

- Decimal numbers are represented using **float** – contain “.” or “e”
- Examples
 - 3.1415
 - -.00134
 - $124e3 = 124 \cdot 10^3$
 - $-2.345e2 = -234.5$
 - $12.3e-4 = 0.00123$
-  Floats are often only approximations, e.g. 0.1 is *not* 1/10
- Extreme values (CPython)
 - `max = 1.7976931348623157e+308`
 - `min = 2.2250738585072014e-308`
- NB: Use module `fractions` for exact fractions/rational numbers.



Python shell

```
> 0.1 + 0.2 + 0.3
| 0.60000000000000001
> (0.1 + 0.2) + 0.3
| 0.60000000000000001
> 0.1 + (0.2 + 0.3)
| 0.6
> type(0.1)
| <class 'float'>
> 1e200 * 1e300
| inf
> 0.1+(0.2+0.3) == (0.1+0.2)+0.3
| False
> x = 0.1 + 0.2
> y = 0.3
> x == y
| False
> print(f'{x:.30f}') # 30 decimals
| 0.30000000000000000044408920985006
> print(f'{y:.30f}') # 30 decimals
| 0.2999999999999999988897769753748
> import sys
> sys.float_info.min
| 2.2250738585072014e-308
> sys.float_info.max
| 1.7976931348623157e+308
```

Associativity rule does
not apply to floats

Question – What addition order is “best”?



- a) $1e10 + 1e-10 + -5e-12 + -1e10$
- b) $1e10 + -1e10 + 1e-10 + -5e-12$
- c) $1e-10 + 1e10 + -1e10 + -5e-12$
- d) $-5e-12 + -1e10 + 1e10 + 1e-10$
- e) Any order is equally good
- f) Don't know

$1e10$	=	100000000000
$-1e10$	=	-100000000000
$1e-10$	=	0.00000000001
$-5e-12$	=	-0.000000000005

Python shell

```
> 1e10 + 1e-10 + -5e-12 + -1e10
| 0.0
> 1e10 + -1e10 + 1e-10 + -5e-12
| 9.50000000000000001e-11
> 1e-10 + 1e10 + -1e10 + -5e-12
| -5e-12
> -5e-12 + -1e10 + 1e10 + 1e-10
| 1e-10
```



a) - d) give four different outputs

Approximating $\pi = 3.14159265359\dots$

$$\frac{\pi^2}{6} = \sum_{k=1}^{+\infty} \frac{1}{k^2} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \dots$$
$$= 1.6449340668\dots$$

Riemann zeta function $\zeta(2)$

pi_approximation_riemann.py

```
apx = 0.0
k = 0.0
while True:
    k = k + 1.0
    apx = apx + 1.0 / (k * k)
    print(k, apx)
```

Output

```
...
94906261.0 1.6449340578345741
94906262.0 1.6449340578345744
94906263.0 1.6449340578345746
94906264.0 1.6449340578345748
94906265.0 1.644934057834575
94906266.0 1.644934057834575
94906267.0 1.644934057834575
94906268.0 1.644934057834575
94906269.0 1.644934057834575
94906270.0 1.644934057834575
...
```



This is not a course in numeric computations – but now you are warned....

Python float \equiv IEEE-754 double precision*

- A binary number is a number in base 2 with digits/bits from {0,1}

$$10110_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 16 + 4 + 2 = 22_{10}$$

- IEEE-754 64-bit double
- | | | |
|-------------------|-------------------------|----------------------------|
| sign s
1 bit | exponent e
11 bits | coefficient c
52 bits |
|-------------------|-------------------------|----------------------------|



Float value	Case
$(-1)^s \cdot (1 + c \cdot 2^{-52}) \cdot 2^{e - 1023}$	$0 < e < 2047$
$(-1)^s \cdot c \cdot 2^{-1074}$	$e = 0, c \neq 0$
+0 and -0	$e = 0, c = 0$
$+\infty$ and $-\infty$	$e = 2047, c = 0$
NaN (“not a number”)	$s = 0, e = 2047, c \neq 0$

```
Python shell
> 1e200 * 1e200
| inf
> -1e200 * 1e200
| -inf
> 1e-200 * 1e-200
| 0.0
> -1e-200 * 1e-200
| -0.0
> 1e200 * 1e200 * 0.0
| nan
```

(*most often, but there is no guarantee given in the Python language specification that floats are represented using IEEE-754)

String literals (type `str`)

- Sequence of characters enclosed by single (') or double (") quotes

```
"a 'quoted' word"      "Hello World"      'abc'
'a "quoted" word'      '_ _ \' _ _'
```

- Escape characters

<code>\n</code>	newline
<code>\t</code>	tab
<code>\\</code>	backslash
<code>\'</code>	single quote
<code>\"</code>	double quote

- A backslash (\) at the end of line, will continue line/string on next line
- Use triple single or double quotes (''' or ''') for enclosing strings spanning more lines
(in particular for Python Docstrings, see [PEP 257](#))

string-test.py


```
print("abc")
print('de\'f')
print("'ghi'")
print("'jk\nl'\\"")
print("mn\
o")
print("p\\q\\tr")
```

Output

```
$ python string-test.py
abc
de'f
'ghi'
'jk
l'"
mno
p\q      r
```

Question – What does the following print ?

```
print ("\\\\" "\n\n")
```

- a) \\\\" "\n\n"
- b) \"\nn"
-  c) \"\n
'
- d) "nn"
- e) \"
'
- f) Don't know

Long string literals

- Long string literals often need to be split over multiple lines
- In Python two (or more) **string literals following each other** will be treated as a single string literal (they can use different quotes)
- Putting **parenthesis** around multiple literals allows line breaks
- Advantages:
 - avoids the backslash at the end of line
 - can use indentation to increase readability
 - allows comments between literals

long-string-literals.py

```
s1 = 'abc' "def" # two string literals
print(s1)
s2 = ''' ''' ''' # avoid escaping quotes
print(s2)
s3 = 'this is a really, really, really, \
really, really, long string'
print(s3)
s4 = ('this is a really, really, '
      'really, really, really, '
      'long string')
print(s4)
very_very_long_variable_name = (
    'this is a really, really, ' # line 1
    'really, really, really, '   # line 2
    "long string"               # line 3
)
print(very_very_long_variable_name)
```

Python shell

```
| abcdef
| '''
| this is a really, really, really, really,
| really, long string
| this is a really, really, really, really,
| really, long string
| this is a really, really, really, really,
| really, long string
```

Raw string literals

- By prefixing a string literal with an `r`, the string literal will be considered a **raw string** and backslashes become literal characters
- Useful in cases where you actually need backslashes in your strings, e.g. when working with Python's regular expression module `re`

Python shell

```
> print('\let\epsilon\varepsilon')      # \v = vertical tab
| \let\epsilon
| arepsilon
> print('\\let\\epsilon\\varepsilon')   # many backslashes
| \let\epsilon\varepsilon
> print(r'\let\epsilon\varepsilon')     # more readable
| \let\epsilon\varepsilon
```

print(...)

- `print` can print zero, one, or more values
- default behavior
 - print a space between values
 - print a line break after printing all values
- default behavior can be changed by **keyword arguments** “`sep`” and “`end`”

Python shell

```
> print()  
|  
> print(7)  
| 7  
> print(2, 'Hello')  
| 2 Hello  
> print(3, 'a', 4)  
| 3 a 4  
> print(3, 'a', 4, sep=':')  
| 3:a:4  
> print(5); print(6)  
| 5  
| 6  
> print(5, end=', '); print(6)  
| 5, 6
```


print(...) and help(...)

Python shell

```
> help(print)
| Help on built-in function print in module builtins:
|
| print(...)
|     print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
|
|     Prints the values to a stream, or to sys.stdout by default.
|     Optional keyword arguments:
|     file: a file-like object (stream); defaults to the current sys.stdout.
|     sep:  string inserted between values, default a space.
|     end:  string appended after the last value, default a newline.
|     flush: whether to forcibly flush the stream.
```

Assignments

- *variable = expression*

`x = 42`

- Multiple assignments – right hand side evaluated before assignment

`x, y, z = 2, 5, 7`

- Useful for swapping

`x, y = y, x`

- Assigning multiple variables same value in left-to-right

`x = y = z = 7`



Warning

`i = 1`

`i = v[i] = 3` # `v[3]` is assigned value 3

In languages like C and C++ instead
`v[1]` is assigned 3

Python is dynamically typed, type(...)

- The current type of a value can be inspected using the **type()** function (that returns a type object)
- In Python the values contained in a variable over time can be of different type
- In languages like C, C++ and Java variables are declared with a given type, e.g.

```
int x = 42;
```

and the different values stored in this variable must remain of this type

Python shell

```
> x = 1
> type(x)
| <class 'int'>
> x = 'Hello'
> type(x)
| <class 'str'>
> type(42)
| <class 'int'>
> type(type(42))
| <class 'type'>
```



x new type

Type conversion


- Convert a value to another type:

new-type(value)

- Sometimes done automatically:

`1.0+7=1.0+float(7)=8.0`

Python shell



```
> float(42)
| 42.0
> int(7.8)
| 7
> x = 7
> print("x = " + x)
| Traceback (most recent call last):
|   File "<stdin>", line 1, in <module>
| TypeError: must be str, not int
> print("x = " + str(x))
| x = 7
> print("x = " + str(float(x)))
| x = 7.0
> int("7.3")
| Traceback (most recent call last):
|   File "<stdin>", line 1, in <module>
| ValueError: invalid literal for int() with base 10: '7.3'
> int(float("7.3"))
| 7
```


Questions – `str(float(int(float("7.5"))))` ?

a) 7

b) 7.0

c) 7.5

d) "7"

 e) "7.0"

f) "7.5"

g) Don't know