# LISTS, DICTIONARIES, FUNCTIONS

Dr Kirill Sidorov
sidorovk@cardiff.ac.uk

School of Computer Science and Informatics
Cardiff University

Like strings, lists are collections of items. Lists can contain values of any type:

```
my_list = [4, False, 'Hello', ['red', 'green']]
```

Some useful operations on lists include concatenation, membership checking, adding/removing/inserting items *etc*...

```
>>> ['red', 'green', 'blue'] + ['cyan', 'magenta']
['red', 'green', 'blue', 'cyan', 'magenta']
>>> 'green' in ['red', 'green', 'blue']
True
>>> len(['red', 'green', 'blue'])
3
```

There are many more, see

https://docs.python.org/3.8/tutorial/datastructures.html

```
cast = {"Spock": "Leonard Nimoy",
        "McCoy": "DeForest Kelley"}

print(cast)
print(cast["McCoy"])
cast["Sulu"] = "George Takei"
print(cast)
print(len(cast))
cast["Spock"] = ["Leonard Nimoy", "Zachary Quinto"]
print(cast)
```

### Example

```
# Iterate over keys in a dictionary
for char in cast:
    print(char + " played by " + cast[char])
```

### Example

```
# Iterate over keys and values in a dictionary
for char, actor in cast:
    print(char + " played by " + actor)
```

Functions name pieces of code the same way variables name values like strings and numbers.

Example

```
def function_name(parameters):
    """Optional documentation."""
    #...
    # Body of the function
    #...
    return value
```

## Example

```
def hello():
    """This function prints a greeting."""
    print("Hello, World!")

hello()
```

## Example

```
def sum(a, b):
    """This function adds two numbers."""
    sum = a + b
    return sum

print(sum(2, 3))
```

```
def sum(a = 1, b = 2):
    """This function adds two numbers.
    By default 1 and 2."""
    print("a is " + str(a) + ", b is " + str(b))
    sum = a + b
    return sum

print(sum())
print(sum(3))
print(sum(3, 4))
print(sum(a = 2, b = 3))
print(sum(b = 4, a = 5))
print(sum(b = 3))
```

A module is a file containing Python definitions and statements. To import a module simply means to make it available (to load the definitions and statements).

## Example

Suppose the function `sum` (above) is defined in file `ex_sum.py`

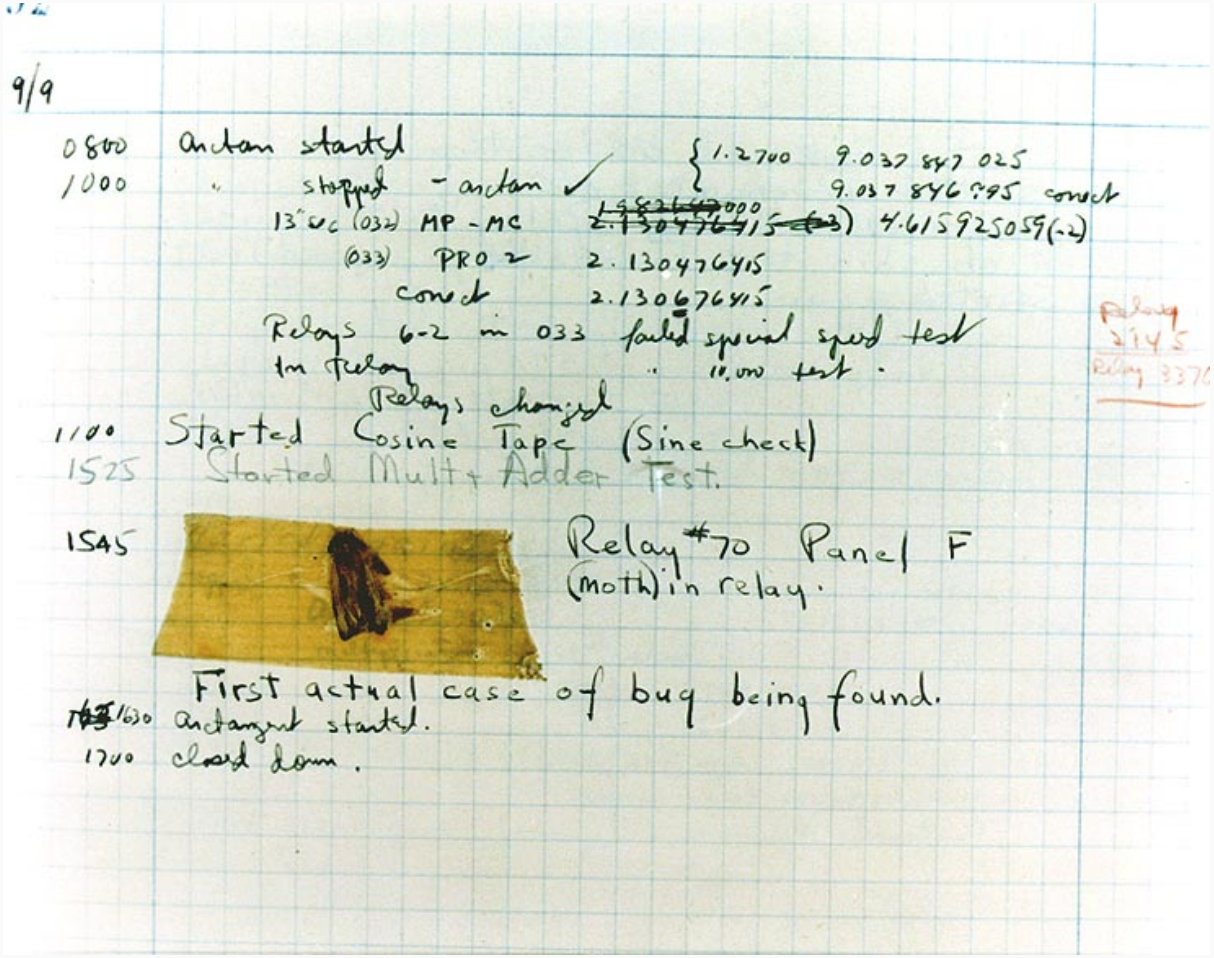
```
>>> import ex_sum
5
>>> ex_sum.sum(3, 4)
7
>>> from ex_sum import sum
>>> sum(5, 6)
11
>>> from ex_sum import *
```

You can use the `doctest` module in Python to easily and automatically test your code.

```
def sum(a, b):
    """This function adds two numbers. E.g.:
    >>> sum(2, 3)
    5
    >>> sum(-1, 1)
    0
    >>> sum(4, 5)
    3
    (This last test is supposed to fail)
    """
    sum = a + b
    return sum
```

To test: `python3 -m doctest -v ex_sum_test.py`

Errors in a program are known as bugs and the art of finding them is called debugging.



# Three types of errors

Three types of errors that you might encounter:

- Syntax errors
- Run-time errors
- Semantic errors

# Syntax errors

- Syntax errors are encountered when you do not follow the rules of the language.
- A piece of program is said to contain a syntax error if it does not conform to the syntax (rules) of the programming language.

## Example

- This is syntactically correct and works:

```
>>> print("Computational Thinking")
Computational Thinking
```

- But the following example contains a syntax error:

```
>>> print)"Computational Thinking")
```

## Example

- This is syntactically correct and works:

```
>>> print("Computational Thinking")
Computational Thinking
```

- But the following example contains a syntax error:

```
>>> print)"Computational Thinking")
  File "<stdin>", line 1
    print)"Computational Thinking")
          ^
SyntaxError: invalid syntax
```

Run-time errors are errors that might occur when a (syntactically correct) program is running, but prescribes Python to do something impossible.

## Examples

- A command says “put coffee into the mug” but you are out of coffee. This is a run-time error. Syntax is correct, but the execution of the program cannot continue.
- Running out of memory.
- Division by zero.
- Accessing an element of a list that does not exist.
- There are countless run-time errors you may encounter...

# Semantic errors

- The program runs without any apparent errors…
- But does not accomplish the task it was intended to do.
- The “meaning” (= semantics) of the program is wrong.
- The human programmer has either devised an incorrect algorithm, or has incorrectly expressed the algorithm in form of a program.

- The program runs without any apparent errors…
- But does not accomplish the task it was intended to do.
- The “meaning” (= semantics) of the program is wrong.
- The human programmer has either devised an incorrect algorithm, or has incorrectly expressed the algorithm in form of a program.

**Example**

You wrote a program to add the numbers $1…10$, but when you run your program it adds the numbers $1…9$.

Q: How do I get rid of bugs in my program?

A: Do not put bugs in your programs in the first place.

Not putting bugs in your code is hard, but it pays off!

Two main principles of any engineering:
ABSTRACTION and COMPOSITION.

- Decompose large problem into small, manageable parts.
  - Each small part (*e.g.* a function) should be simple enough for you to completely understand how it works.
  - Test each small part until it is rock-solid.
- Compose the large solution out of well-tested solutions to sub-problems.
  - Use contracts to define how small parts fit together.
- This decomposition may span multiple levels.

- Get rid of mutable state where possible.
  - Use pure functions (that just compute and return a value, but do not change anything).
  - Avoid unnecessary global variables. Keep mutable state in one place.
  - Avoid "leaky abstractions". A function may have local mutable state, as long as this is not visible from outside.
- Write less code and write smarter.
  - Whenever you write complicated code you open your self up to bugs. Keep things simple and short.
- Write abstractly. Instead of solving a problem, solve a family of problems in the neighborhood of the problem.
  - Small changes to the problem should result in small changes in the solution.

## From Python style guide:

- Use 4-space indentation, and no tabs.
- Use blank lines to separate functions and larger blocks of code inside functions.
- When possible, put comments on a line of their own.
- Use spaces around operators and after commas, but not directly inside bracketing constructs:
  `a = f(1, 2) + g(3, 4)`
- Name your functions consistently; the convention is to use `lower_case_with_underscores` for functions and methods.

## There is much more — read the style guide