CM1101 Computational Thinking

Introduction to Python

Dr Kirill Sidorov
SidorovK@cardiff.ac.uk
www.facebook.com/kirill.sidorov

School of Computer Science and Informatics
Cardiff University, UK

Comments

- Comments allow you to document what your program does.
- This becomes even more important as your programs get larger and more complicated.
- Comments are inserted using the hash symbol: #
    - The rest of the line after the # is simply ignored by the interpreter — so, you can put any text in the comments.

The `if` statement

```
if condition:
    # This is executed if 'condition' is True
```

A more complete variant:

```
if condition:
    # This is executed if 'condition' is True
else:
    # This is executed otherwise
```

Indentation is important!

The `if` statement

```
if condition:
    # This is executed if 'condition' is True
elif other_condition:
    # Otherwise, this is executed
    # (if 'other_condition' is True)
else:
    # This is executed otherwise
```

The `if` statement

Example:

```
number = float(input("Enter a number: "))
if number > 0:
    print("It is positive.")
# Nothing happens otherwise
```

Example:

```
number = float(input("Enter a number: "))
if number > 0:
    print("It is positive.")
else:
    print("It is not positive.")
```

The `if` statement

```python
if pints > 2:
    print("You cannot drive a car!")
    if pints > 6:
        print("Call a cab!")
    else:
        print("Ride your bicycle!")
elif pints > 0:
    print("Drive very cautiously!")
else:
    print("It is ok to drive!")
```

The `while` loop

```python
while condition:
    # Keep doing this while 'condition' is True
```

Example:

```python
a = 0
while a < 10:
    a = a + 1
    print(a)
```

The `for` loop

```
for x in iterable:
    # Do something with x as it
    # goes (iterates) over iterable
```

Example:

```
s = "Hello"    # We want to iterate over this string
for ch in s:
    print(ch)
```

The `for` loop

Example:

```
for x in range(0, 5):
    print(x*x)
```

Example:

```
for x in range(4, 12, 2):
    for power in [2, 3, 6]:
        print(str(x) + " to the power " +
              str(power) + " is " + str(x ** power))
```

Style

From Python style guide:

- Use 4-space indentation, and no tabs.
- Use blank lines to separate functions and classes, 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 more — read the style guide

Functions

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

Syntax:

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

Functions

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))
```

Functions

Example:

```python
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))
```

Modules

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 *
```

Testing

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

```
def sum(a, b):
    """This function adds two numbers. For example:
    >>> 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`

Dictionaries

```
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)
```

Dictionaries

Example:

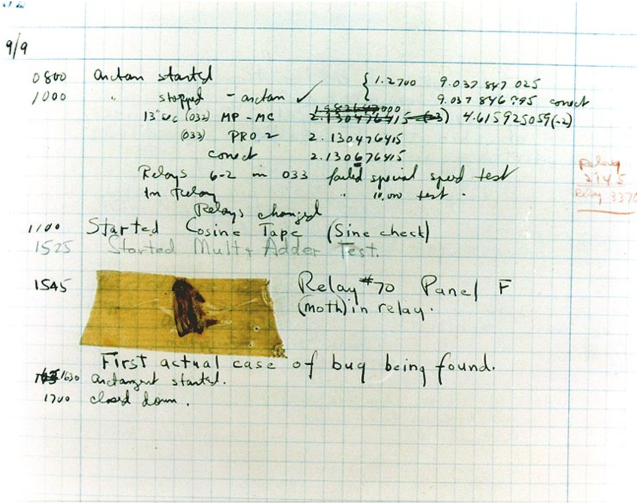
```
# Iterate over keys in a dictionary
for char in cast:
    print(char + " played by " + cast[char])
```

Example:

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

Dealing with errors

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



9/9

0800 antan started {1.2700 9.037 847 025

1000 " stopped - antan ✓ 9.037 846 995 conect

13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415 (-3)~~ 4.615925059(-2)

(033) PRO 2 2.130476415

conect 2.130676415

Relays 6-2 in 033 failed special speed test

In relay " 10,000 test .

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Mult + Adder Test.

1545 Relay #70 Panel F (moth) in relay.

First actual case of bug being found.

1630 antangent started.

1700 closed down.

Relay 2145

Relay 3370

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.

Syntax errors

Example:

- This is syntactically correct and works:

  ```
  >>> print("Kirill Sidorov")

  Kirill Sidorov
  ```

- But the following example contains a syntax error:

  ```
  >>> print)"Kirill Sidorov")

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

Run-time errors

- As the name suggests, these are errors that might occur when a (syntactically correct) program is running.
- You are unlikely to encounter run-time errors in your early days of learning to program in Python.

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.
- Out of memory.
- Division by zero.
- *etc.*

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 has incorrectly expressed the procedure (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.

Debugging

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

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

Some tips for writing correct code

Not putting bugs in your code is harder, 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.

Some tips for writing correct code

- Get rid of mutable state where possible.
    - Use pure functions (that just compute and return a value, but do not change any thing).
    - 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.