Basics of Python

ENGF0002: Design and Professional Skills

Prof. Mark Handley
University College London, UK

Term 1, 2020

Algorithm of the Week

Euclid's algorithm for computing the GCD. (I)

Euclid of Alexandria (300 BC) was interested in properties of integer and rational numbers.

Finding the **greatest common denominator** (GCD) allowed them to simplify fractions:

Eg. $\frac{390253}{228769} = \frac{29 \cdot 13457}{17 \cdot 13457} = \frac{29}{17}$ since $GCD(390253, 228769) = 13457$.

But how to compute the GCD function without resorting to factoring the numbers (which turns out to be a hard problem for large numbers)?

Euclid's algorithm for computing the GCD. (II)

Euclid's algorithm for computing the GCD (in prose)

- Consider the two integers for which you want to find the GCD.
- Take the smaller one, and subtract it from the larger one.
- Replace the larger one with this result, and keep the smaller one, as your two new numbers.
- Repeat until the two numbers are equal.
- Once they are, the resulting (equal) numbers are the GCD.

Let's build a Python program where we execute the concrete steps of the algorithm for the sample numbers 42 and 30.

Why Euclid's algorithm works? (I)

Consider the positive integers a and b:

$$a = \alpha \cdot gcd(a, b), \tag{1}$$
$$b = \beta \cdot gcd(a, b), \tag{2}$$

where $gcd(\alpha, \beta) = 1$.
Without loss of generality, consider $a > b$. After a single step:

$$a' = a - b \tag{3}$$
$$= (\alpha - \beta) \cdot gcd(a, b). \tag{4}$$

Note that a' is always strictly positive.

Why Euclid's algorithm works? (II)

We can show that $gcd(a', b) = gcd(a, b)$.
Proof by contradiction:

- We know that gcd(a,b) is a factor of both a' and b.
- So if $gcd(a', b) > gcd(a, b)$ it must be that: $gcd(\alpha - \beta, \beta) = c > 1$.
- Thus we can rewrite $\beta = r \cdot c$ and $\alpha - \beta = r' \cdot c \Leftrightarrow \alpha = (r' + r) \cdot c$.
- However that means that α and β are both divisible by $c > 1$. Which contradicts $gcd(\alpha, \beta) = 1$.

Thus by repeatedly applying the steps of the algorithm we keep the gcd in each step constant while reducing the size of *a* and *b*. Until they are equal to the gcd.

Calculations following Euclid’s algorithm for GCD.

```python
# Step by step GCD computation for numbers 42 (=6*7) and 30 (=6*5)
a = 42
b = 30

# First, a > b, pick a
v = a - b      # v == 12
a = v

# Now, b > a, pick b
b = b - a      # b == 18

# Still, b > a, pick b
b = b - a      # b == 6

# Now, a > b, pick a
a = a - b      # a == 6

print(a)       # Now a == b so, we stop. GCD is in a, or, b.
```

Assignments, values & expressions.

Let's understand all aspects of the Python code in this example.

- **Statements** perform an action within the program. In this example all code lines are statements.
- **Assignments** are statements that assign an expression (right hand side of =) to a variable with a number in the program (left hand side of =). Eg.

  ```
  a = 42
  ```

  assigns 42 to the named variable `a`.
- **Expressions** are fragments of code that return a computed quantity.
- **Values** are expressions that return a constant, eg. `42` and `30`.
- More generally **arithmetic expressions** such as `b - a` perform a computation and return its result. They can be nested arbitrarily.

Some Python idioms.

- The order of evaluation of arithmetic expressions follows the **usual precedence rules** of integer arithmetic, eg. `2+3*4` is computed as `2+(3*4) = 14`. You may force a different order of evaluation by using parenthesis, eg. `(2+3)*4 = 20`.
- Python allows for multiple **simultaneous assignments**. Eg. lines 2–3 could be expressed as:

  ```
  a, b = 42, 30
  ```

  All expressions on the right hand-side are first evaluated before any assignment occurs.
- Python allows for **reusing names** in the left-hand side of an assignment as in line 10.

  ```
  b = b - a
  ```

  Only **reuse variable names if they represent the same** high-level concept.

Types & Programming Languages. (I)

Python is a strongly typed and dynamically typed language.

- **Strongly typed**: Every variable, or expression has a **type** associated with it. All expressions in our example program are of type **int** representing integers. Only operations supported by this type can be performed on variables or expressions of that type.
- **Dynamically typed**: While the program runs the types of objects are tracked, and operations are checked before being applied to ensure they are permitted.

In contract **statically typed** languages run this check before the program executes. However, to decide **type-safety** before execution they **lose expressiveness** or require **type annotations**.

Types & Programming Languages. (II)

Types and formal methods

Type systems are the most popular **formal methods** that aid program correctness. They ensure only permissible operations are executed, and eliminate a large number of potential trivial bugs. But are **not a substitute for testing** or further **verification** for correctness.

The Python `int` integer type

The type **int** is primitive, in that the Python language itself knows about it out of the box. It can represent **integers of arbitrary length** (unlike other languages), and supports:

- The arithmetic operators $+$, $-$, $*$ and $**$ (to the power) return integers.
- The integer division $//$ and remainder operator % also return integers.
- Division $/$, but does not return an integer!
- The function `int(x)` returns the integer representation of `x` (if it exists).

Primitive and user-defined types

Python supports out-of-the-box a number of types, including integers, complex numbers, real numbers (`float`), truth values (`bool`), strings of characters (`str`), and many data structures.

A **full list of all supported operations** can be found in the documentation for all primitive types. **Study it** carefully. `https://docs.python.org/3/library/stdtypes.html`.

User-defined types and abstraction

High-level programming languages allow programmers to define their own types, associated with their own data and permitted operations. This is a key feature that makes programming in such languages expressive—and closer to the intent of the programmer—and free of the need to reflect in very low-level types, minimizing mistakes.

Controlling the execution of programs.

So far we have used Python as a **glorified calculator**.

We had to interpret ourselves the algorithmic commands relating to doing **different operations depending on conditions**, and **repeating operations while a condition holds**.

- **Conditional execution** of commands can be achieved by using the `if ... then ...` control structure taking the form:

  ```
  if condition:
      block of statements 1.
  else:
      block of statements 2.
  ```

- **Repeated execution** while a condition holds may be achieved thorugh the `while ...` control structure taking the form:

  ```
  while condition:
      block of statements 1.
  ```

Automatic program control for Euclid’s algorithm.

Refactoring the euclid algorithm to take advantage of `while` and `if` control structures.

```python
# Step by step GCD computation for numbers 42 (=6*7) and 30 (=6*5)
a = 42
b = 30

while a != b:
    if a > b:
        a = a - b
    else:
        b = b - a

print(a)       # Now a == b so, we stop. GCD is in a, or, b.
```

Conditions and the `bool` type.

What is the nature of conditions in Python programs?

- Conditions are just **expressions**, that return a result of type **bool**.
- A variable of type `bool` takes one of two values, **True** or **False**.
- A number of arithmetic operators return a boolean, including:

  == for equality

  < less than

  > greater than

  `!`= not equal, etc.
- Booleans may be linked together through '**and**', '**or**' and '**not**'.
- Those **lazily** evaluate the expressions until its value is determined.

Loops, and the `while` control structure.

```
5   while a != b:
6       if a > b:
7           a = a - b
8       else:
9           b = b - a
```

Keep re-executing a block until a condition stops holding.

- Upon encountering a `while` the program **evaluates** the condition.
- If the condition is **True** it starts executing the statements in the block.
- When the block ends, it **continues** back to the `while` (line 5).
- if **False** it skips the block and **breaks** to line 10.
- You can explicitly use `continue` and `break` in the program.

Conditional execution with the `if` control structure.

```
6    if a > b:
7        a = a - b
8    else:
9        b = b - a
```

Execute a block, or another, depending on a condition.

- Upon encountering an `if` the program **evaluates** the condition.
- If **True** it executes the first block (line 7).
- Else, if **False**, it executes the second block (line 9)

How are blocks defined?

In Python blocks are defined as a set of statements with the same degree of indentation (whitespace). Use 3 or 4 spaces to group blocks together, and do not use any tabs.

Euclid as a function.

Refactor Euclid’s algorithm as a function with two parameters.

```python
def GCD(a,b):
    """Compute the GCD of two positive int."""
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a

ax = 42
bx = 30
result = GCD(ax, bx) # Call the GCD function
print(result)
```

Re-using code through functions. (I)

Functions allow us to **define blocks of code**, and **reuse** them without copying them.

```
def GCD(a,b):
```

- Declare a function through the keyword `def` followed by a **function name**, and a list of **named parameters** in brackets.
- The names parameters are **available within the block** (body) of the function, but not outside (**variable scope**).
- The **return** statement **exits** the function, and may **return a value**.

```
    return a
```

Re-using code through functions (II).

You may call a function, pass to it different parameters, and get the result.

```python
result = GCD(ax, bx) # Call the GCD function
```

- You may **apply** any parameters to a function to **call** it.
- **Parameters are expressions** that are evaluated before the function is called.
- The function call itself is an expression that results in the **value returned** or **None**.

Functions are key to supporting **abstraction** in programs.

The ‘scope’ of variables.

Why Scope variables?

Scope ensures that the names of variables from different parts of the program do not clash and confuse the programmer, by ensuring a degree of ‘locality’.

LEGB Rule defines when variables are available to a block (scope):

- **L, Local**: Names assigned in any way within a function.
- **E, Enclosing-function locals**: Name in the local scope of any and all statically enclosing functions, from inner to outer.
- **G, Global (module)**: Names assigned at the top-level of a module file, or by executing a global statement.
- **B, Built-in (Python)**: Names in the built-in names module.

The Abstraction and DRY principles.

Benjamin C. Pierce in *Types and Programming Languages* (2002):

The Abstraction Principle

Each significant piece of functionality in a program should be implemented in **just one place** in the source code. Where similar functions are carried out by distinct pieces of code, it is **generally beneficial** to combine them into one by **abstracting out** the varying parts.

Andy Hunt and Dave Thomas in *The Pragmatic Programmer* (1999)

Don’t Repeat Yourself (DRY) Principle

Every piece of knowledge must have a **single, unambiguous, authoritative representation** within a system.

Euclid as a function and unit test

Refactor the Euclid Example with `ax,bx = 42,30` as a test.

```python
def GCD(a,b):
    """Compute the GCD of two positive int."""
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a

# Test case for function
def test_euclid():
    ax = 42                 # Create test case
    bx = 30                 # from simple example.
    r = GCD(ax,bx)          # Turn test print,
    assert r == 6           # into assert.
```

What makes a good test?

Brian W. Kernighan and Rob Pike in *The Practice of programming* (1999), provide some advice:

- **Coverage**, make sure all lines, and code paths are tested.
- **Compare** different implementations to check refactoring and refinement.
- Test for **boundary conditions**, values and special cases.
- Identify and test that **pre- and post- conditions** hold.
- Identify and check for **conservation properties**.
- Check **error** returns and exceptional paths.

Tests cannot check all possible paths of large programs, but **model checkers** can be used to test paths more efficiently.

The ‘happy path’, errors and exceptions.

Users of your code will use it in **invalid ways**, and **exceptional circumstances** will occur.

Handling errors and exceptions:

- Upon an error code should **fail fast**, and **never fail silently**.
- Your **code should handle errors and exceptions** that are the result of external factors.
- An external factor is **your future self**, using your own code in an unexpected manner.
- Handling exceptional paths must not obscure the intent of your program (the **happy path**).
- Use **language facilities** to handle errors and exceptions.

Raising an exception upon error.

Refactor Euclid to ensure that inputs are positive integers.

```
def GCD(a,b):
    """Compute the GCD of two positive int."""
    if not (a > 0 and b > 0):
        raise ArithmeticError("%s, %s: Must be positive int." % (a,b))
    while (a != b):
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

The **raise** keyword **interrupts the flow of the program**, and exits the block with an exception immediately.

The `try / except / finally` control structure.

To **handle an exception raised** by a block of code we use:

```
try:
    Block that may `raise' exception ExceptionName.
except ExceptionName as VariableName:
    Block that handles the exception.
finally:
    Block that is always executed.
```

- The program executes the block in **try** until the block ends.
- If the block raises an exception, the execution ends, and it is matched to the exception name in the **except** handler.
- If the exception matches the block of the exception handler is executed, otherwise the exception is raised further.
- In all cases the **finally** block of code is executed.

How to test & handle exceptions.

A test that ensures the exception handler is activated for invalid input.

```
def test_euclid_exc():
    try:
        r = GCD(5, -1)  # tiggers exception.
        assert False    # skip rest of block.
    except ArithmeticError as e:
        # Exec. except block for `Exception'
        assert "Must be positive int." in str(e)
    except:
        assert False    # skip other blocks.
    finally:
        assert True     # Always execute finally.
```

Note that multiple, including generic, **except** blocks may be present.

Detect errors at a low level, handle them at a high level.

The Python raise mechanism interrupts the flow, and bubbles up the exception to outer calling blocks. Why?

- Errors are often detected **deep within programs**.
  Eg. a file name does not exist when you are about to open it.
- At such a low-level, it is **not known how to handle** them.
- Thus error handling happens at a higher level,
  when **options for recovery** are known.
- Use the **finally** for local clean-up upon an error.
- Do not use exceptions as part of the happy path, and vice versa.

Only **handle errors at a level where you know how to recover**, otherwise raise them up further.

The practice of refactoring.

Programs are living things that **evolve**.

- Programming is a **dynamic activity**: a program is **re-written and re-tested** many times per minute.
- Evolving the program to increase its quality is called **refactoring**.
- **Good design** (abstraction and DRY) & the existence of **good tests** ensure refactoring does **not introduce new errors** in existing code.

Version Control enables fearless refactoring

What if refactoring breaks your working program? Tests will detect this, but how can you go back to the working previous version? The solution is to use a version control system, such as `git`, which we will study later.