From problems to programs

Michel Wermelinger (michel.wermelinger@open.ac.uk)

The most difficult part is problem-solving, not coding. Fortunately, there are recurring problem types with boilerplate solution templates. The process becomes:

1. Recognise the type of problem
2. Get the corresponding solution pattern
3. Instantiate the pattern to get an algorithm for the problem at hand
4. Translate (largely 'automatically') the algorithm to code

Patterns and algorithms are in plain English (with variables in italics and numbered lists of steps), not pseudo-code. Formatting conventions (indent lists, start with a colon) map directly to Python. Abstracting is difficult: we provide the abstractions (patterns), students make them concrete (algorithms). Here are some examples adapted from The Open University's *Introduction to Computing and IT 2* (http://www.open.ac.uk/courses/modules/tm112).

Formula problems

Type: calculate output values directly from input values, e.g. convert units, compute BMI
Pattern (uses sequence of instructions only):

1. initialise the *input variables*
2. set the *output variable* to the value of the formula applied to the *input variables*
3. print the *output* variable

Example algorithm:

1. initialise the *weight* in kilograms
2. initialise the *height* in meters
3. set *BMI* to *weight* / (*height* × *height*)
4. print *BMI*

one step in pattern may become several steps in algorithm

Program: initialise *x* –> `x = int(input("message "))`

Case analysis

Type: the input values fall into one of multiple exclusive cases, e.g. BMI category
Pattern (uses selection):

1. initialise the *input* variables
2. if *input* values fall into the first case:
    a. compute *outputs* according to the first case
3. otherwise if *inputs* fall into the second case:
    a. compute *outputs* according to the second case
4. etc.
5. otherwise:
    a. compute *outputs* according to the last case
6. print the *outputs*

Example algorithm:

1. initialise the *BMI* in kilograms per square meters
2. if *BMI* is below 18.5:
    a. set *category* to 'underweight'
3. otherwise if *BMI* is below 25:
    a. set *category* to 'normal'
4. otherwise if *BMI* is below 30:
    a. set *category* to 'overweight'
5. otherwise:
    a. set *category* to 'obese'
6. print *category*

Program: otherwise if –> `elif`; otherwise –> `else`

Reduce

Type: 'summarise' a list (array) of values as a single value, e.g. sum or length of a list
Pattern (uses iteration):

1. initialise the input *list*
2. initialise the *summary* with a suitable value
3. for each *item* in the *list*:
    a. if the *item* satisfies a certain condition:
        i. update the *summary* according to the *item*

final print step omitted due to limited space

Sum algorithm:

1. initialise a list of *numbers*
2. set the *sum* to zero
3. for each *number* in *numbers*:
    a. increment the *sum* by *number*

some steps in the pattern may not be needed

Length algorithm: increment always by 1
Program: for each … in the … –> `for … in …`

Search

Type: find a value of the list that satisfies a certain condition, e.g. find a negative number
Pattern:

1. initialise the input *list*
2. set *found* to the null value
3. for each *item* in the *list*:
    a. if the *item* satisfies a certain condition:
        i. set *found* to the *item*

search reduces a list to one value so this is an instance of the previous pattern

Program: null value –> `None`

Transform

Type: transform all items that satisfy a condition, e.g. convert Celsius values to Fahrenheit
Pattern:

1. initialise the *input list*
2. initialise the *output list* to the empty list
3. for each *input value* of the *input list*:
    a. if the *input value* satisfies a certain condition:
        i. compute the *output value* from the *input value*
        ii. append the *output value* to the *output list*

Program: empty list –> `[]`; *multiple words* –> `multiple_words`;
append *item* to *list* –> `list.append(item)` or `list = list + [item]`

Filter

Type: special case of search (find <u>all</u> values satisfying a condition)
Pattern: as for Transform, but without step 3.a.i and with 3.a.ii appending *input value* to *output list*

Concluding Remarks

The common structure of patterns shows how problems are similar or related, and shows some of the roles variables play (container – the list, iterator – each item, accumulator – the resulting value or list). Patterns are a thinking scaffold that guides the algorithm development: What variables do I need? What are their the initial values? How do I update them?

More complex problems can be handled by decomposing them into simpler existing problem types and putting the corresponding patterns together, e.g. the mean of a numeric list is two reductions (sum and length) followed by a formula (sum over length).