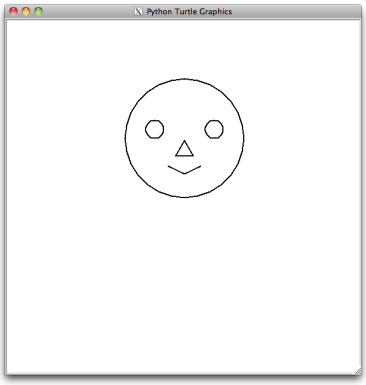
Computer Science 1
Fancy Faces — Lecture Part 1

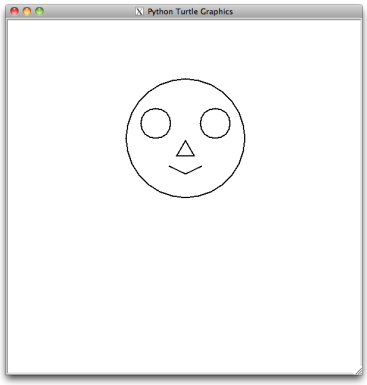
08/16/2018

Problem

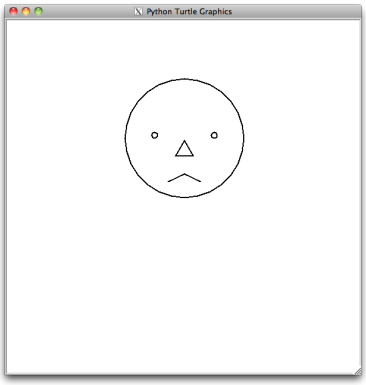
After the last job's face-drawing triumph, ACME Stick Figure Company has contracted with us to produce faces for their upcoming line of stick figures. While marketing confirms that the "traditional" happy face has broad appeal, select segments of the target demographic might prefer larger or smaller eyes and might prefer a frown. To keep production costs down, they need a *general* program that can produce a variety of faces, rather than writing a *specific* program for each face.



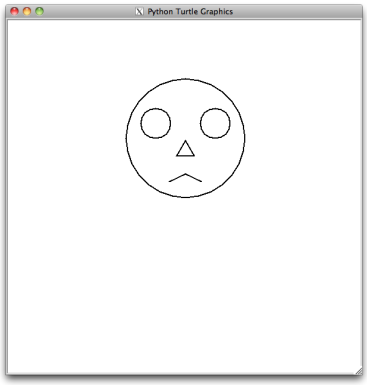
`mouth_type == "smile"`
`eye_radius == 15`



`mouth_type == "smile"`
`eye_radius == 25`



`mouth_type == "frown"`
`eye_radius == 5`



`mouth_type == "frown"`
`eye_radius == 25`

Investigations

We investigate to study the problem and consider what capabilities we would need to solve it effectively.

First, what are the similarities and differences among the different faces?

- similar: shape and size of the border (head)
- similar: location of the center of the mouth
- similar: location, shape, and size of the nose
- similar: size of the eyes and location of their bottom tangent
- different: type of mouth ("up" for smile; "down" for frown)
- different: size (radius) of each eye

The similarities suggest that a small collection of procedures can be reused to produce a variety of faces. The differences suggest *parameterizing* some of these procedures by *arguments* in order to draw the specific features.

For example, the previous solution had a procedure to draw an eye. In this solution, we will continue to have a procedure to draw an eye, but that procedure will take an argument specifying the radius of the eye to draw. We have already used many procedures that take arguments (e.g., `turtle.circle`, `turtle.forward`); now we will define our own procedures that take arguments.

Parameters and Arguments

To handle these differences, we investigate how to write procedures that have parameters. We can pass a value to a procedure, if we add a *parameter* inside the () parentheses of the definition.

For example, we might write the eye drawing procedure in Python like this:

```
def draw_eye( eye_radius ):
    """
    Draws a single eye as an 'eye_radius' circle.

    eye_radius -- PositiveInteger. determines the radius of the eyes

    pre-conditions: turtle is facing East at the bottom of the eye location.
    post-conditions: turtle is in the same state as at the start.
    """

    turtle.down()
    turtle.circle( eye_radius )
    turtle.up()
```

Notice that we document the procedure's parameters in the *docstring* that goes inside the definition. The triple-quote syntax ensures that the `pydoc` program will be able to extract this documentation and make it publicly available outside the program itself. The documentation also specifies the conditions that code should expect both before and after execution.

The definition of the `draw_eye` procedure has a *parameter* named `eye_radius` inside the () parentheses of the definition header. The parameter is a 'placeholder' for a value that the program must pass in as an *argument* at runtime. The procedure's *parameter has a value*

only when called at runtime. If a procedure has one parameter, the caller must pass a value, and we call that value the *argument*.

We might call our procedure like this to draw an eye with radius 45.

```
draw_eye( 45 )
```

This call *binds the value* 45 *to the parameter name* `eye_radius`, and the code executes by drawing a circle of radius 45. The argument value is substituted for all occurrences of the parameter in the procedure definition.

Binding Inputs to Names and Converting Strings

Next we investigate how to get an input value and convert it to an appropriate type.

Now that we see how to define a procedure that has a parameter, we need to capture the input from the user. The `input` function outputs the message given as its argument to the user, and it waits to receive a response typed by the user and terminated by pressing the Return/Enter key.

Here is an example of interactive use; the user entered the bold text:

```
>>> input( "What is your name? ") )
What is your name? Fred
'Fred'
>>>
```

The above output in single quotes[1] is a *character string*, or simply a string, and it is an instance of the Python **str** data type. The input function will *return the response typed by the user as a string*, and the program can use that as the answer.

Perhaps we might call the eye drawing procedure like this:

```
draw_eye( input( "What size eye? ") )
```

Anything we type in response to `input( "What size eye? ")` would become the value for `eye_radius`. If we typed *big* and hit Enter in response to the prompt `What size eye?`, then the procedure `draw_eye()` would receive a *string* `"big"` as the argument value for the `draw_eye` parameter.

The problem is that the `turtle.circle` function expects a number for its argument type, and a string is not a number. The missing step is the conversion of the input string into a number. In Python, the `int` function performs this *casting*. Here is the correction:

```
draw_eye( int( input( "What size eye? ") ) )
```

What happens when we call `draw_eye` after answering `"big"`? Python detects an error because the `int` function cannot convert `"big"` to an integral number. Should we do something to correct for this? No. The problem is with the user because they entered the wrong kind of value. They should have read the documentation which specified that `eye_radius` must be an integer.

---

1. In Python, we can represent a string in single or double quotes. The interpreter always uses single quotes.

If we call `draw_eye` after answering `"27"` to the prompt, then the `int` function converts the string `"27"` to an integer value, and execution passes the integer into `draw_eye`, which in turn passes the value to the `turtle.circle` function.

From Input Values to Arguments

After we get an input and convert it to the right type, we have to pass those values as arguments to a procedure.

We can write a procedure with a *comma-separated list* of parameters to pass one or more argument values to a procedure at runtime. Since we need two values from the user and the eye radius needs conversion to an *integer* for the eye radius, there are several steps before we can call the primary drawing procedure.

Because there are several steps, it is helpful to break things down and save values by associating or binding the values to names that we can use in later steps.

The assignment operator, =, performs this binding, and we say that this *sets the value of the name*. The syntax is `<name> = <value>`[2], where the `<name>` must be a legal, single word name for a thing, and `<value>` is the value *to set*.

We can prompt the user for the mouth type and bind the *string* value returned by `input` to a name that we will pass later to the drawing procedure. Then we can convert the user input for the radius into an *integer* for the eye radius, and bind that value to another name.

Putting everything together regarding input and conversion, we can write this code for most of the top level of the program:

```
mouth_type = input( "Enter mouth type (\"smile\" or \"frown\"): " )

eye_radius = int( input( "Enter eye radius (a positive integer): " ) )

# pass the bound values in as arguments to another procedure

init_world_and_draw_fancy_face( mouth_type, eye_radius )
```

Conditional Statements and Relational/Logical Expressions

The problem's differences suggest that we need to *choose* to execute different code based on different input values.

For example, our previous solution had a procedure to draw a smiling mouth. In our new solution, we will continue to have a procedure to draw a smiling mouth, but that procedure will need to have a parameter for the type so that another procedure can supply an argument

---

2. The angle brackets indicate these are *metalanguage* placeholders; substitute Python content to create an actual statement.

specifying the mouth type (`"smile"` or `"frown"`). The body of this procedure will choose to execute one block[3] of code when the mouth type is `"smile"`, another block of code when the mouth type is `"frown"`, and yet another block of code when the mouth type is anything else.

This is a job for conditional `if statements`, which we will need for drawing the different mouth types.

Conditional statements begin with an `if` line containing a *test expression*, followed by a *colon* (`:`) and a 'then clause' (*an indented block of statements to execute when the test expression is True*). This sequence is possibly followed by zero or more `elif` *tests and clauses*, and possibly a final `else` clause. The entire `if-elif-else` conditional statement defines a set of *alternative execution paths* through the code. The path of execution chosen is based on evaluation of the test written on the `if` line and on any `elif` lines that may be present.

Tests are expressions that evaluate to `True` or `False`, and these words must be capitalized in Python. The values form a *data type* called **Boolean**, and the actual type name is **bool** in Python. The name comes from George Boole, the pioneer in the algebra of logic.

For our problem, we need to choose whether to execute code for a frown, a smile, or something else based on the value of the argument passed in through the mouth type parameter.

We write that in Python like this:

```
def draw_mouth( mouth_type ):

    if mouth_type == "frown":
        # code to draw the frown
    elif mouth_type == "smile":
        # code to draw the smile
    else:
        # code to draw a mouth neither smiling nor frowning

    # ... code executing after the if statement ...
```

As an example, when called with a `mouth_type` matching `'smile'`, the execution skips the first if, matches the second check, and executes the code for the `elif`.

Syntax and Semantics

The general form of a Python `if` statement is as follows:

```
if Bool-Test-Expression0:
    statements to execute if Bool-Test-Expression0 is True
elif Bool-Test-Expression1:
    statements to execute if Bool-Test-Expression1 is True
elif Bool-Test-Expression2:
    statements to execute if Bool-Test-Expression2 is True
:
```

3. Sequences of code that execute as a unit are called *blocks*.

```
:
elif Bool-Test-ExpressionN:
    statements to execute if Bool-Test-ExpressionN is True
else:
    statements to execute if none of the Bool-Test-Expressions are True
```

Each expression evaluates to `True` or `False` at runtime, and the Python interpreter evaluates the sequence in *top-down* order.

We don't write any `elif` clauses if the problem does not require any. If there are none, it is just a two-choice scenario based on a single test.

The `else` clause is also optional. Without an `else` clause, if the tests in the if statement produce a value of False, then none of the code in the `then` or `elif` parts of the `if` statement executes.

Note: After statements in one clause of an `if` block are executed, no further tests are evaluated, and no other statements in the `if` statement execute. The execution continues at the next statement after the `if` block.

Relational and Logical Operators

The relational operators check whether or not the values of the expressions on their left and right sides 'match' each other according to the comparison specified by the operator.

These are the relational operators in Python:

1. `==`, and `!=` are "equal to" and "not equal to";
2. `<`, and `<=` are "less than" and "less than or equal to:"; and
3. `>`, and `>=` are "greater than" and "greater than or equal to".

There are also Boolean, logical operators that work on Boolean expressions.

1. "`Left-Expr` **and** `Right-Expr`" is `True` only if both `Left-Expr` and `Right-Expr` are `True`.
2. "`Left-Expr` **or** `Right-Expr`" is `True` if either `Left-Expr` or `Right-Expr` is `True`.
3. "**not** `expr`" is `True` only if `expr` is `False`.

For our problem, we want to make sure that the eye radius does not get too small (i.e. less than 0) or too large to fit into the head. In other words, draw the figure only if the eye radius is *between 0 and 35 inclusive* ($0 <= radius <= 35$). (These values were empirically derived.)

Here is a code fragment showing how we can code this check:

```
if eye_radius >= 0 and eye_radius <= 35:

    init_world_and_draw_fancy_face( mouth_type, eye_radius )
else:
    # code cannot draw the face because of invalid input
```

Design

Now that we have investigated the problem to discover and learn the tools needed, we start the high level design that identifies the parts of a solution.

Solution Components

We will partition our solution into several algorithms that break the program down into procedures, and create an outline of how they will flow.

- Procedure to gather information before drawing
  - prompt the user to input the mouth type
  - prompt the user to input the eye radius and convert to number
  - check the inputs for correct ranges (and stop if incorrect)
  - initialize and draw the face using the given mouth type and eye radius
  - tell the user to hit *Enter* when done viewing the face
  - close the drawing canvas
- Procedure to draw the whole face with parameters `mouth_type` and `eye_radius`
  - initialize the drawing canvas
  - draw the border
  - draw the mouth using the given mouth type
  - draw the nose
  - draw the eyes using the given radius
- Procedure to draw the border (head)
  - Draw a circle with radius 100
- Procedure to draw the mouth with a parameter to specify `mouth_type`
  - If `mouth_type` is "smile" draw a smile
  - Otherwise if `mouth_type` is "frown" draw a frown
  - Otherwise draw a "tight-lipped" mouth
- Procedure to draw the nose
  - move to the center of the face
  - draw an equilateral triangle
  - move back to the bottom of face (the starting point)
- Procedure to draw the eyes with a parameter to specify `eye_radius`
  - move to the bottom of one eye
  - call `draw_eye` procedure with the eye radius
  - move to the bottom of the other eye
  - call `draw_eye` procedure with the eye radius
  - move to the bottom of the face
- Procedure to draw an eye with a parameter to specify `eye_radius`

Stubbing a Solution

One technique to evolve the outline involves writing *stub procedures* for the parts and connecting them together by having them call each other. These *stubs* will form the basis for the implementation of the actual solution when they are filled in with code. Below, this example stub shows how we can write one.

```
def draw_eye( eye_radius ):
    """
    note: a docstring body makes this a legal, 'do-nothing' stub procedure.
    """
```

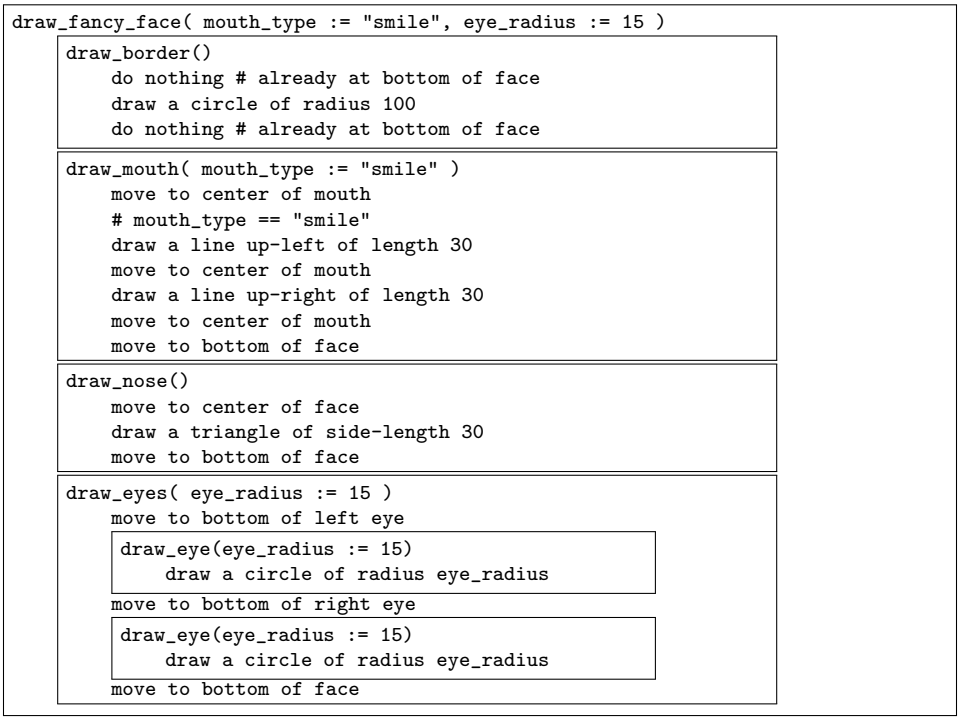
While our design appears complete, it misses the need to test how the program works when given a variety of different inputs for the mouth and eye. For that we add a stub that will run multiple tests. Below is a list of stubs for the program.

- `draw_border()`
- `draw_mouth( mouth_type )`
- `draw_nose()`
- `draw_eyes( eye_radius )`
- `draw_eye( eye_radius )`
- `draw_fancy_face( mouth_type, eye_radius )`
- `init_world_and_draw_fancy_face( mouth_type, eye_radius, message="")`
- `run_test_cases()`
- `prompt_and_draw_fancy_face()`

Execution Diagram

We want to check the design by *executing on paper*. For this we can draw pictures to trace through what the computer would do to execute a program that follows this plan.

We can visualize the execution of the `draw_fancy_face` procedure with an *execution diagram*. This picture shows that, when one procedure calls another procedure, the *calling procedure* remains active until the *callee* finishes executing.



Code Implementation

See the accompanying file `fancy_face.py` for a solution that also includes *test procedures*.

Execution: Testing (Test Cases, Procedures, etc.)

We should be sure that our program is able to produce the variety of faces from the first page, and we should also test our program with several different argument values.

Here is an outline of some tests that `run_test_cases` does:

- `mouth_type == "smile"`, `eye_radius == 15`
- `mouth_type == "smile"`, `eye_radius == 25`
- `mouth_type == "frown"`, `eye_radius == 5`
- `mouth_type == "frown"`, `eye_radius == 25`
- `mouth_type == "smile"`, `eye_radius == 35`
- `mouth_type == "frown"`, `eye_radius == 15`
- `mouth_type == "zzz"`, `eye_radius == 10`
- `mouth_type = "smile"`, `eye_radius = -1`
- `mouth_type == "zzz"`, `eye_radius == 100`

Although we won't require programs to gracefully handle bad input at this stage, it helps to see how the program behaves in response to bad, unexpected or invalid input:

- `mouth_type = "smile"`, `eye_radius = "ten"`

Note that the bad input test will cause the Python interpreter to fail; we gave a string type for a value whose type should be a number.

Assessment: How good is our solution?

We can run it in an *ad hoc*, interactive fashion to produce one image.

Or we can run the program test procedure to see that it works in the desired ways.

At this point, we should always assess our work by asking ourselves questions like these:

- What works well?
- What works poorly?
- How could the code be improved?
- What missing conditions or situations were identified in testing?
- What improvements would make the code smaller, faster, more reliable?