Think Python 2e, Chapter 4 Notes

Geoffrey Matthews

September 13, 2022

The turtle module

```
import turtle
bob = turtle.Turtle()
turtle.mainloop()
```

- This will create a single turtle, `bob`
- `bob` will react to commands entered in the shell
- Try:

```
bob.fd(100)
bob.rt(120)
bob.fd(100)
bob.rt(120)
bob.fd(100)
```

Simple looping

What do you think this does?

```
for i in range(4):
    print('Hello!')
```

Simple looping

What do you think this does?

```python
import turtle
bob = turtle.Turtle()

for i in range(4):
    bob.fd(100)
    bob.lt(90)
```

Encapsulation

Write a **function** that takes a turtle as argument, and makes it draw a square.

Encapsulation

Write a **function** that takes a turtle as argument, and makes it draw a square.

```
def square(t):
    for i in range(4):
        t.fd(100)
        t.lt(90)
```

```
square(bob)
square(alice)
```

- Also called **Abstraction**: giving a name to something.
- Makes it very easy to reuse code in different contexts.

Generalization

Add a parameter `length` to `square`

```
def square(t):
    for i in range(4):
        t.fd(100)
        t.lt(90)
```

```
square(bob)
square(alice)
```

Generalization

Add a parameter `length` to `square`

```
def square(t):
    for i in range(4):
        t.fd(100)
        t.lt(90)
```

```
square(bob)
square(alice)
```

```
def square(t, length):
    for i in range(4):
        t.fd(length)
        t.lt(90)
```

```
square(bob, 100)
square(alice, 200)
```

Further Generalization

Change `square` to `polygon`.

```
def square(t, length):
    for i in range(4):
        t.fd(length)
        t.lt(90)
```

```
square(bob, 100)
square(alice, 200)
```

Further Generalization

Change `square` to `polygon`.

```python
def square(t, length):
    for i in range(4):
        t.fd(length)
        t.lt(90)
```

```python
square(bob, 100)
square(alice, 200)
```

```python
def polygon(t, n, length):
    angle = 360/n
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

```python
polygon(bob, 5, 100)
polygon(alice, 8, 200)
```

Keyword arguments

```python
def polygon(t, n, length):
    angle = 360/n
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

```python
polygon(bob, 5, 100)
polygon(alice, 8, 200)
```

```python
polygon(bob, n=5, length=100)
polygon(alice, length=200, n=8)
```

Interface design

```
def circle(t, r):
    circumference = 2*math.pi*r
    n = max(int(circumference/5), 5)
    polygon(t, n, circumference/n)
```

- We want to draw a *smooth* circle using a polygon with many sides.
- How many sides do we use?
- This should *not* be part of the interface.
- It is part of the *implementation*, but should not concern the user.
- We find a number so that the straight lines are not more than 5 pixels in length.

Refactoring

We want to write `arc` that will draw part of a circle.

```
def polygon(t, n, length):
    angle = 360/n
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

```
def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = max(int(arc_length / 3), 1)
    step_length = arc_length / n
    step_angle = angle / n
    for i in range(n):
        t.fd(step_length)
        t.lt(step_angle)
```

The last part of this function looks like `polygon` but we can't reuse `polygon` since it assumes we want 360°

Refactoring

Old definition:

```
def polygon(t, n, length):
    angle = 360/n
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

New definition:

```
def polyline(t, n, length, angle):
    for i in range(n):
        t.fd(length)
        t.lt(angle)
def polygon(t, n, length):
    angle = 360.0 / n
    polyline(t, n, length, angle)
```

We broke one function into two, like factoring $15 = 3 \cdot 5$

Refactoring

```
def polyline(t, n, length, angle):
    for i in range(n):
        t.fd(length)
        t.lt(angle)
def polygon(t, n, length):
    angle = 360.0 / n
    polyline(t, n, length, angle)
```

New version is more useful:

```
def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(t, n, step_length, step_angle)
def circle(t, r):
    arc(t, r, 360)
```

A development plan

1. Start by writing a small program with no function definitions.
2. Once you get the program working, identify a coherent piece of it, encapsulate the piece in a function and give it a name.
3. Generalize the function by adding appropriate parameters.
4. Repeat steps 1–3 until you have a set of working functions.
5. Look for opportunities to improve the program by refactoring.
   - For example, if you have similar code in several places, consider factoring it into an appropriately general function.

Docstrings

```
def polyline(t, n, length, angle):
    """Draws n line segments with the given length and
    angle (in degrees) between them.  t is a turtle.
    """
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

```
help(polyline)
Help on function polyline in module __main__:

polyline(t, n, length, angle)
    Draws n line segments with the given length and
    angle (in degrees) between them.  t is a turtle.
```

Documentation and debugging

- Functions expect their arguments to meet certain conditions.
  - For example, `polyline` requires four arguments: `t` has to be a Turtle; `n` has to be an integer; `length` should be a positive number; and `angle` has to be a number, which is understood to be in degrees.
- These expectations are called the **preconditions**
- If the preconditions are met, the function should guarantee that certain other conditions are also met.
  - For example `polyline` will draw line segments of the right length and angle.
- These expectactions are called the **postconditions**

Documentation and debugging

- If your program has a bug, and the preconditions for a function are not met, the bug is probably in the code that calls the function.
- If the preconditions are met, but the postconditions are not, then the bug is in the function.
- If the preconditions and postconditions of a function are clear, they can help with debugging.

Vocabulary

method: A function that is associated with an object and called using dot notation.

loop: A part of a program that can run repeatedly.

encapsulation: The process of transforming a sequence of statements into a function definition.

generalization: The process of replacing something unnecessarily specific (like a number) with something appropriately general (like a variable or parameter).

keyword argument: An argument that includes the name of the parameter as a "keyword".

interface: A description of how to use a function, including the name and descriptions of the arguments and return value.

Vocabulary

refactoring: The process of modifying a working program to improve function interfaces and other qualities of the code.

development plan: A process for writing programs.

docstring: A string that appears at the top of a function definition to document the function's interface.

precondition: A requirement that should be satisfied by the caller before a function starts.

postcondition: A requirement that should be satisfied by the function before it ends.