# Turtle Python 4 – Animation and User Input

Although not a fundamental part of Computer Science, *it is great fun to write programs that are animated and interactive*, for example video games! To make this achievable as quickly as possible, here we are already – in only the third Turtle Python document – learning how to do so. Some of the material in §4 below (handling keyboard and mouse input) might seem rather technical. But if you find it difficult, feel free to ignore it until you come to want to do something that requires it. *None of this material is essential from the point of view of learning Computer Science*, and *much of this is specific to the Turtle System* (because the standard "Core" of Python has very limited facilities for graphics and input). But it covers things that you might well want to do to make your programs interactive and entertaining.

## 1. Introduction to Animated Movement

Simulating a moving object on a screen assumes that we are keeping track of where it is at each "instant". Suppose that the object has already been drawn at some initial position. Then visual movement is achieved with a "loop" that cycles through the following operations repeatedly:

```
    Erase the object's image at its current position
    Update the object's position
    Draw the object in its new position
    Pause for a time, depending on the desired speed of movement
```

Usually the object's position will be stored as x- and y-coordinates, and movement will be expressed in terms of *velocities* in the x- (horizontal) and y- (vertical) directions. Suppose, for example, that we start off with the object 100 units from the left of the canvas and 700 units from the top (so x-coordinate 100 and y-coordinate 700); we refer to this as position "(100,700)". (Note that in computer graphics, the y-axis points *downwards* rather than upwards as in conventional mathematics.) Then suppose we want the object to move to position (900,300) over 100 steps, so that the x-coordinate (let's call this "x") increases by 8 each step, and the y-coordinate (call this "y") decreases by 4 each time. Our loop now looks something like this:

```
Start by making x equal to 100, and y equal to 700
Repeat 100 times:
    Erase the object's image at position (x,y)
    Add 8 to x; subtract 4 from y
    Draw the object at position (x,y)
    Pause for a time, depending on the desired speed of movement
```

Suppose that our "object" is a red ball of radius 50. Then to draw it at position (x,y), we simply move the *Turtle* to that position using the command `setxy(x,y)` and draw a red blot of radius 50 there. To erase it, we draw a *white* blot in the same position. If we put these commands within the loop structure above, that should work, but the movement is likely to appear "flickery" because the object is being repeatedly erased and drawn again. To deal with this problem, the display can be "frozen" (by preventing screen updates) before the erasing takes place, and then "unfrozen" (by re-enabling screen updates) after the object has been redrawn. This makes the redrawing almost simultaneous with the erasing, so the object won't seem to disappear at all:

```
x = 100
y = 700
for count in range(100):
    noupdate()
```

```
        colour(white)
        blot(50)
        x = x+8
        y = y-4
        setxy(x,y)
        colour(red)
        blot(50)
        update()
        pause(10)
```

This is, in fact, *almost* identical to the Example program "Moving ball (using variables)" within Examples menu 4. The only very slight difference in that program is use of `blot(51)` rather than `blot(50)` to erase the ball – this is because if *Turtle* is run in the online web browser version, an image-processing technique called *anti-aliasing* "smudges" the edges of the blot, so that the red colour extends slightly beyond the strict boundary, requiring a larger white blot to erase it.

## 2. Acceleration and Bouncing

In the example above, the red "ball" moves a constant amount between loop cycles: its x-coordinate increases by 8, and its y-coordinate decreases by 4. So the ball has a velocity in the x-direction of +8 units per cycle, and a velocity in the y-direction of -4 units per cycle. To make the example more easily flexible, we could create variables *xVelocity* and *yVelocity* to represent these velocities. Then these two assignments would be added to the program above:

```
xVelocity = 8
yVelocity = -4
```

while the looping changes to the x- and y-coordinates would become:

```
x = x+xVelocity
y = y+yVelocity
```

Having done this, we can now accelerate (or decelerate) the ball by changing the values of *xVelocity* and *yVelocity* – for example, if *xVelocity* is made equal to 16, the horizontal movement will be twice as fast, and if *yVelocity* is made equal to 0, the ball will stop rising and instead move only horizontally.

We can also make the ball "bounce" by inverting the velocities when the ball overlaps the relevant edge of the canvas (e.g. by changing *xVelocity* from +8 to -8, or *yVelocity* from -4 to +4). Since the ball has a radius of 50 units, and the default canvas coordinates range from 0 to 999 inclusive, this bouncing should be made to happen when either *x* or *y* is less than 50, or greater than 949. We achieve this using two simple "if" statements, as follows:

```
if (x<50) or (x>949):
    xVelocity = -xVelocity
if (y<50) or (y>949):
    yVelocity = -yVelocity
```

Now instead of limiting the ball's movement to 100 steps (using the "for" loop), we can allow it to continue indefinitely, knowing that it will never leave the canvas because it will always bounce back from the edges. One way of doing this is to put the movement into a "while" loop whose condition – here "0<1" – remains true eternally:

```
while 0<1:
    ...
```

The complete program can be found in Examples menu 5, under the title "Bouncing ball (using variables)". (As before, this draws a white blot radius 51, to deal with web browser anti-aliasing.)

## 3. Taking Advantage of the Turtle Coordinates

In the examples above, we used variables *x* and *y* to record the current coordinates of the moving object. But if we are dealing with just one object, then we can instead take advantage of the *Turtle*'s inbuilt variables *turtx* and *turty* to do this work for us. If we draw the object only at the position of the *Turtle*, and accordingly move the *Turtle* where we want the object to go, then *turtx* and *turty* will automatically be updated as we move around, enabling our program to be shorter. The original commands:

```
x = 100
y = 700
```

can then be replaced by:

```
setxy(100,700)
```

which sets *turtx* to 100, and *turty* to 700. In a similar way, the three commands:

```
x = x+8
y = y-4
setxy(x,y)
```

can be replaced by the single command:

```
movexy(8,-4)
```

which adds 8 to the *Turtle*'s x-coordinate *turtx*, subtracts 4 from the *Turtle*'s y-coordinate *turty*, and thus moves the *Turtle* to that position for the ball to be drawn there (note that `movexy` differs from `drawxy`, which draws a line as the *turtle* moves but is otherwise similar).

To see completed programs using this technique, go to Examples menu 4 where you will find "Moving ball (using Turtle)" and "Bouncing ball (using Turtle)".

## 4. Mouse and Keyboard Input

Let us now see how user input can be incorporated into running programs, since this can be used quite generally to make your programs more interesting (e.g. by incorporating real-time user controls). *Turtle* provides quite powerful facilities for input, falling into three main categories, concerned with capturing:

- Keyboard typed input
- Key presses
- Mouse movements and mouse clicks

Note the distinction between the first two of these: one involves the *characters* that are typed in from the keyboard (taking account of lower/upper case etc.), while the other involves the *physical key presses*. When typing takes place, typically *both* types of input are recorded simultaneously.

### 4.1 Keyboard Typed Input

When keys are pressed while a program is running within the *Turtle System*, the typed characters are standardly put into a *keyboard buffer* which stores them for later reading. The virtue of maintaining such a buffer is that you don't need to write code to read characters one by one as they are typed, which would be tricky and cumbersome. The keyboard buffer has a default length of 32, meaning that if you type 32 characters without reading any of them in the meantime, then no more characters will be accepted (and the machine will emit a beep as you type them). The integer function `keystatus(\keybuffer)` can be used to discover how many characters are currently in the keyboard buffer, while the procedure `reset(\keybuffer)` empties the buffer completely. If you need a larger buffer, to accommodate, say, up to 100 characters, then you can change its size using the command `keybuffer(100)`.

Core Python provides only one function for reading keyboard input, which first displays a prompt and then reads an entire line (i.e. it requires *Return* or *Enter* to been pressed). Suppose, for example, that you want the user to input their name, then you might use the instruction:

```
s = input("What is your name? ")
```

This displays "What is your name?" followed by a space, and waits for the user to type a line of text (ending with the *Return* or *Enter* key), then assigns that line of text to the variable s – which is thus a *string variable* (i.e. it holds a *string of characters* rather than a *number*). For a simple program that uses this technique, see "Asking for typed input" in Examples menu 5. If you want to read a line of input without giving any prompt, you can either give an empty prompt:

```
s = input("")
```

or none at all:

```
s = input().
```

(Note that although *Turtle*'s default keyboard buffer holds only 32 characters, the `input` command can cope with more, because it continuously reads characters from the keyboard buffer – thus freeing up space for more characters – until it encounters an end-of-line character; but the end-of-line character itself is discarded and does not become part of the string variable's value. In *Turtle*'s default state, a string variable can hold up to 64 characters.)

*Turtle* provides another function for reading *characters* (rather than whole lines) from the keyboard buffer. Suppose, for example, that you want to read *up to 10 characters*, then you could write:

```
s = read(10)
```

This instruction reads up to 10 characters from the keyboard buffer (thus removing them from the buffer and freeing up space in it), and assigns them to the relevant string variable `s`. Thus if the user previously typed only "Hello", then after the command executes, the variable's value will be "Hello" and the keyboard buffer will be empty, but if the user previously typed "Hello there, how are you?", then the variable's value will now be "Hello ther" (10 characters, including a space) and the keyboard buffer will be left containing "e, how are you?". To read the contents of the keyboard buffer *without removing any characters from it*, use `s = read(0)`. (The name of the `read` function echoes the Python equivalent, `sys.stdin.read`, which requires importing the `sys` module and so is not part of the Core language. *Turtle Python* is designed to be self-contained, and so does not require any additional modules.)

## 4.2 Key Presses

While any program is running within *Turtle*, details of *the latest key press* are recorded continuously, and this information can be consulted using the special "query" functions `?key` and `?kshift`, both of which return integer values. Thus for example:

```
n = ?key
```

will make n equal to the relevant key value. If the `A` key is currently being pressed, then n will be made equal to 65 (the ASCII code for 'A'); but if the `A` key has just been released (with no other key being pressed in the meantime), then `?key` will have been negated so that n will be -65. In this way, `?key` can be used both to identify the last key press, and also whether that press is still continuing. To make identification easier, *Turtle* provides built-in *keycode* constants, including `\a` (65) to `\z` (90), likewise for the digit keys (e.g. `\4`, or `\#4` on a numeric keypad) and other standard character keys (e.g. `\=`), and for the special keys: `\alt`, `\backspace`, `\capslock`, `\ctrl`, `\delete`, `\down`, `\end`, `\enter`, `\escape`, `\f1` (etc.), `\home`, `\insert`, `\left`, `\lwin`, `\pgdn`, `\pgup`, `\return`, `\right`, `\rwin`, `\shift`, `\space`, `\tab`, and `\up`. Thus to do something repeatedly until the *ESCAPE* key has been pressed – irrespective of whether that key is then immediately released – it is simplest to use a loop citing `\escape` (actual value 27), but also incorporating the `abs` function (meaning absolute value, e.g. `abs(-27)` is equal to 27):

```
while abs(?key) != \escape:   # in Python, "!=" means "is not equal to"
    <do something>
```

Whenever a key is pressed, its "shift status" is recorded as a single number, which is calculated as 128 plus 8 if the *Shift* key was being held down at the time, plus 16 if *Alt* was held down, plus 32 if *Ctrl* was held down. Thus if the last key was pressed with *Ctrl* and *Alt* both held down, then `?kshift` will return the value 128+16+32 = 176 (at least while the key is still pressed; again this goes negative, to -176, if called after the key has been released). This value is also recorded separately for each key, so that `keystatus(\a)`, for example, will return the corresponding value for the last press of the `A` key (even if other keys have been pressed in the meantime). Much as with the keyboard buffer (in the previous section), `reset(\a)` puts the *keystatus* of the `A` key back to its default setting (here -1). More detail on all of this is given in the built-in "User Input" help panel (within *Turtle*'s "QuickHelp 1" tab), which gives a handy reference.

## 4.3 Mouse Movements and Mouse Clicks

While a program is running, *Turtle* continuously keeps track of the position of the mouse over the Canvas, with the x- and y-coordinates being given by the integer functions `?mousex` and `?mousey` respectively. Likewise, when a mouse click takes place, the x- and y-coordinates of the click position are given by the integer functions `?clickx` and `?clicky`. Rather like the function `?kshift` (which, as we saw in the previous section, records the status of the last key press), so `?click` records the status of the last mouse click, calculated as 128 plus 1 for a left-click, 2 for a right-click, 4 for a middle-click, 8 if *Shift* was held down, 16 if *Alt* was held down, 32 if *Ctrl* was held down, and 64 if it was a double-click (and again as with key presses, `?click` is made negative when the click event finishes). But often a more convenient way of checking for a specific mouse click is to use the functions `?lmouse`, `?rmouse` and `?mmouse`, which return the status for the latest click with the corresponding mouse button (left, right, and middle respectively). Thus, for example, a loop like this will wait until the left mouse button *is actually being clicked*:

```
while (?lmouse<=0):   # in Python, "<=" means "is less than or equal to"
    # do nothing
```

If, on the other hand, you want a loop to terminate *after* the left mouse button has been clicked, whether or not it is still being held down, you could use:

```
reset(?lmouse)
while abs(?lmouse<=128):
    <do something>
```

The command `reset(?lmouse)` sets the left-mouse status to -1. When that mouse button is clicked, the status value will change to 129 (assuming an ordinary click with no *Shift* etc.), and when it is released, to -129. Hence looping until it achieves an absolute value greater than 128 does the trick.

## 4.4 Detecting Timed Input

*Turtle* also provides a useful function for detecting either keyboard presses or mouse input over a specific time period. Suppose, for example, that you want to give the user up to 3 seconds to press the *ENTER* (or *RETURN*) key, then you could use the instruction:

```
n = detect(\enter,3000)
```

This waits for 3000 milliseconds, unless the *ENTER* key is pressed meanwhile. If *ENTER* is pressed in time, then n will be assigned the appropriate status code, as explained in §4.2 above (i.e. 128, or 136 if *Shift* was being held down, etc.). If *ENTER* has not been detected after 3 seconds, then n is made equal to 0. If you want *Turtle* to wait indefinitely for input, then specify a time parameter of 0.

Note that while *Turtle* is waiting for input here, anything typed on the keyboard will be "echoed" on the Console. To control whether such echoing occurs, you can use the `keyecho` instruction, for example:

```
print('Press ENTER within 3 seconds')
keyecho(False)
n = detect(\enter,3000)
if n == 0:
    print('Too late!')
else:
    print('Well done!')
keyecho(True)
```

in which the final instruction turns the keyboard echo back on (assuming that's wanted).

Exactly the same technique applies for mouse clicks, except that here the first parameter to the `detect` function – called an *inputcode* – will be one of the *mousecodes* `\lmouse`, `\rmouse` or `\mmouse`, depending on whether you are waiting for a left, right, or middle mouse click. And in this case, the appropriate status code returned by the function will be as explained in §4.3 above.

There is also one special *inputcode* which handles both key presses and mouse clicks, and thus enables you to write a program that waits for whichever input occurs first. Thus, for example,

```
n = detect(\mousekey,1500)
```

will wait for up to 1.5 seconds until either a key is pressed or a mouse button is clicked. If neither happens, then n will be made equal to 0. If a mouse click occurs, then n will be made equal to 1, 2 or 4 respectively, depending on whether the click involves the left, right, or middle button. If a key is pressed, then n will be made equal to the relevant *keycode* (e.g. 13 = `\return` = `\enter` for the *RETURN* or *ENTER* key; 27 = `\escape` for the *ESCAPE* key; 65 = \a for the `A` key, etc.).

## 4.5 Examples

To see a wide range of these techniques in action, take a look at the programs in Examples menu 5, "user input, interaction, and games":

- "Asking for typed input" illustrates simple use of the `input` command (as we saw in §4.1).
- "Mouse reaction game" illustrates waiting for the *ESCAPE* key, waiting for a left mouse click, and identifying the colour of the pixel at the mouse position.
- "Typing test (checking characters)" uses `detect(\escape,5000)` to implement a wait of up to 5 seconds, and `detect(\keybuffer,0)` to wait until a character appears in the keyboard buffer.
- "Typing test (checking keys)" uses `?key` and `?kshift`, as well as `detect(\key,0)` – note here that every *querycode* (such as "?key") has a corresponding *inputcode* (such as "\key").
- "Iteration game (Collatz sequence)" displays *on the Canvas* (rather than the Console) a number being typed in. This is tricky, because the user might want to edit the number as they go (i.e. using the backspace key to delete digits), which the keyboard buffer can handle, but that means that the number has to be repeatedly redrawn (rather than just being extended). So the box command is used to clear the relevant area, and `read(0)` to show the contents of the keyboard buffer.
- "Throwing sponges at a moving face" and "Arcade shooting game" detect left mouse clicks and mouse coordinates.
- "Colouring cells" uses `detect(\mousekey,5000)` to be able to identify different kinds of mouse clicks, while also waiting for the *ESCAPE* key.
- "Snake (classic game)" detects different key presses to determine how the snake should move.
- "Drawing to the mouse" detects mouse clicks and locations, setting colour according to the horizontal position of the click.
- "Painting application" detects mouse clicks and locations, setting brush width according to the horizontal position of the click, and colour according to the pixel where the click occurs.