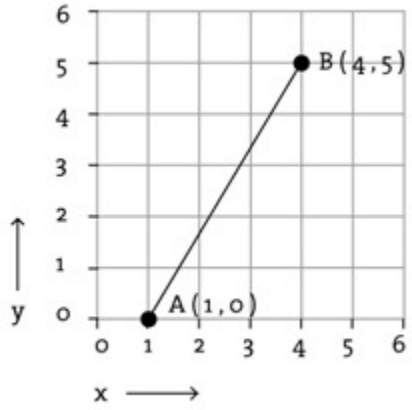
Chapter 1

Coordinate System

Coordinate System and Shapes

Coordinate Space

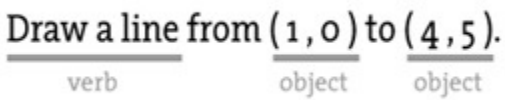
Before we begin programming with Processing, we must first channel our eighth grade selves, pull out a piece of graph paper, and draw a line. The shortest distance between two points is a good old fashioned line, and this is where we begin, with two points on that graph paper.



The above figure shows a line between point A (1,0) and point B (4,5). If you wanted to direct a friend of yours to draw that same line, you would give them a shout and say "draw a line from the point one-zero to the point four-five, please." Well, for the moment, imagine your friend was a computer and you wanted to instruct this digital pal to display that same line on its screen. The same command applies (only this time you can skip the pleasantries and you will be required to employ a precise formatting). Here, the instruction will look like this:

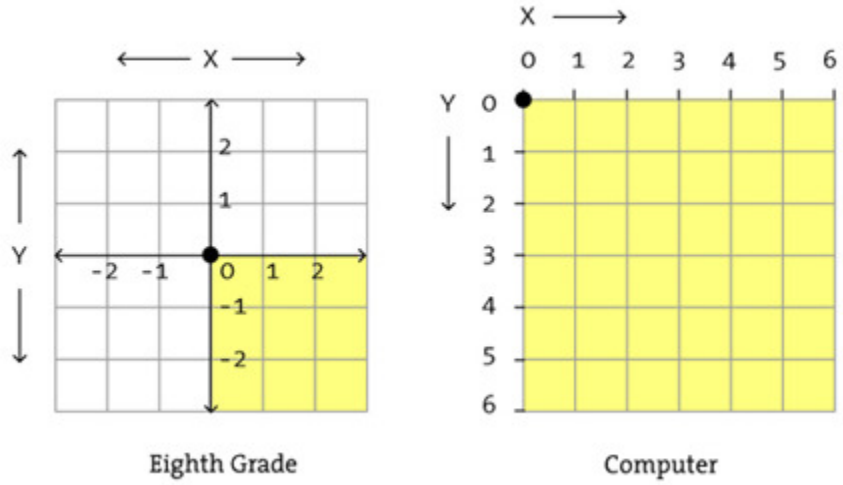
```
line(1,0,4,5);
```

Even without having studied the syntax of writing code, the above statement should make a fair amount of sense. We are providing a *command* (which we will refer to as a "function") for the machine to follow entitled "line." In addition, we are specifying some arguments for how that line should be drawn, from point A (1,0) to point B (4,5). If you think of that line of code as a sentence, the function is a verb and the arguments are the objects of the sentence. The code sentence also ends with a semicolon instead of a period.



The key here is to realize that the computer screen is nothing more than a fancier piece of graph paper. Each pixel of the screen is a coordinate - two numbers, an "x" (horizontal) and a "y" (vertical) - that determines the location of a point in space. And it is our job to specify what shapes and colors should appear at these pixel coordinates.

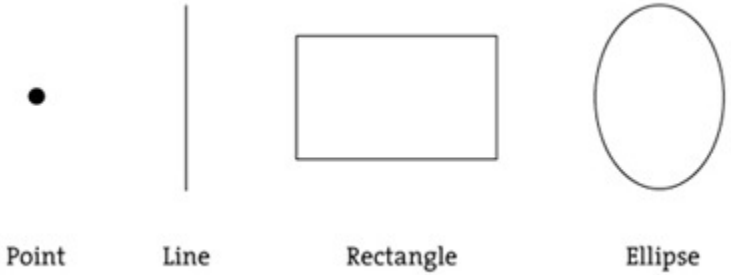
Nevertheless, there is a catch here. The graph paper from eighth grade ("Cartesian coordinate system") placed (0,0) in the center with the y-axis pointing up and the x-axis pointing to the right (in the positive direction, negative down and to the left). The coordinate system for pixels in a computer window, however, is reversed along the y-axis. (0,0) can be found at the top left with the positive direction to the right horizontally and down vertically.



Simple Shapes

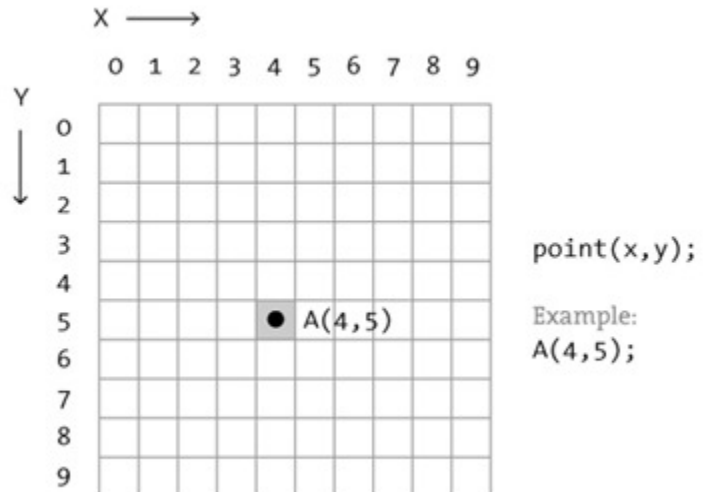
Simple Shapes

The vast majority of the programming examples you'll see with Processing are visual in nature. These examples, at their core, involve drawing shapes and setting pixels. Let's begin by looking at four primitive shapes.

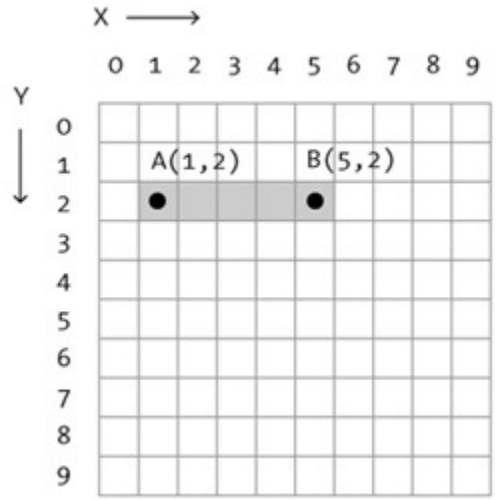


For each shape, we will ask ourselves what information is required to specify the location and size (and later color) of that shape and learn how Processing expects to receive that information. In each of the diagrams below, we'll assume a window with a width of 10 pixels and height of 10 pixels. This isn't particularly realistic since when you really start coding you will most likely work with much larger windows (10x10 pixels is barely a few millimeters of screen space.) Nevertheless for demonstration purposes, it is nice to work with smaller numbers in order to present the pixels as they might appear on graph paper (for now) to better illustrate the inner workings of each line of code.

A **point()** is the easiest of the shapes and a good place to start. To draw a point, we only need an x and y coordinate.



A line() isn't terribly difficult either and simply requires two points: (x1,y1) and (x2,y2):

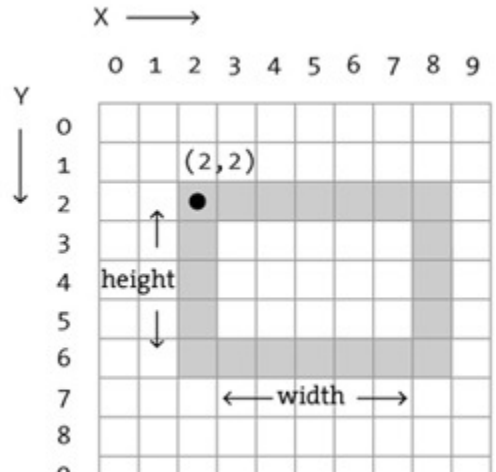



```
line(x1,y1,x2,y2);
     Point A Point B
```

Example:

```
line(1,2,5,2);
```

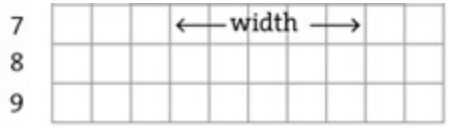
Once we arrive at drawing a rect(), things become a bit more complicated. In Processing, a rectangle is specified by the coordinate for the top left corner of the rectangle, as well as its width and height.



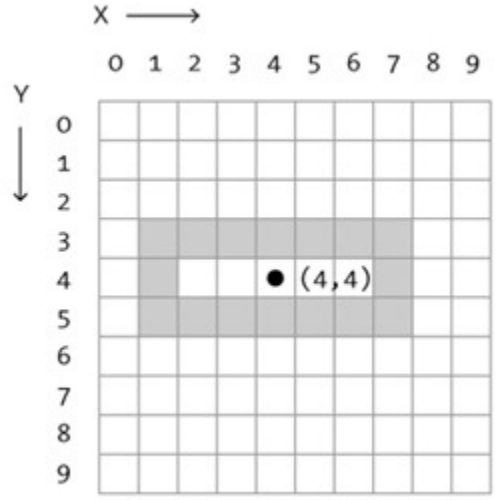

```
rect(x,y,width,height);
```

Example:

```
rect(2,2,7,5);
```

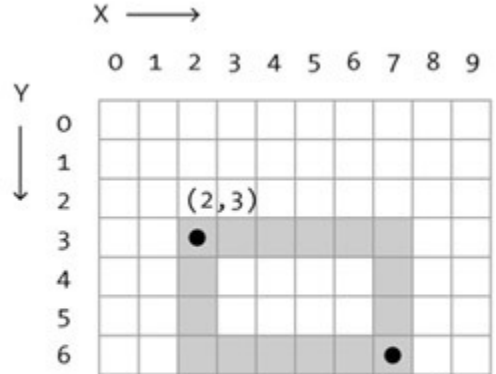


A second way to draw a rectangle involves specifying the centerpoint, along with width and height. If we prefer this method, we first indicate that we want to use the "CENTER" mode before the instruction for the rectangle itself. Note that Processing is case-sensitive.

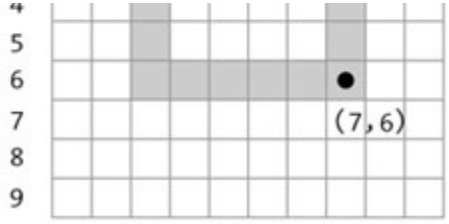



```
rectMode(CENTER);
rect(x,y,width,height);

Example:
rectMode(CENTER);
rect(4,4,7,3);
```

Finally, we can also draw a rectangle with two points (the top left corner and the bottom right corner). The mode here is "CORNERS".



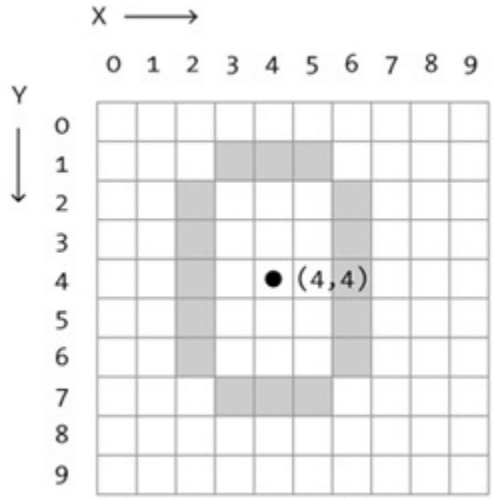

```
rectMode(CORNERS);
rect(x1,y1,x2,y2);

Example:
rectMode(CORNERS);
```



Example:

```
rectMode(CORNERS);
rect(2,3,7,6);
```

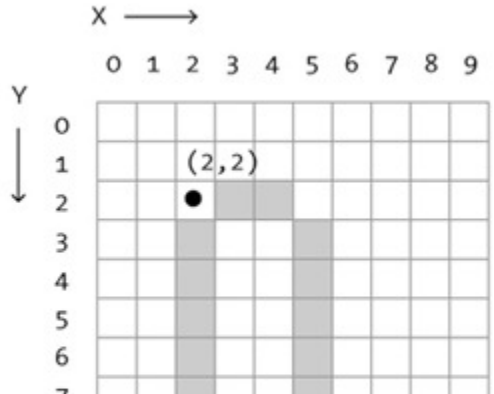
Once we have become comfortable with the concept of drawing a rectangle, an **ellipse()** is a snap. In fact, it is identical to **rect()** with the difference being that an ellipse is drawn where the bounding box of the rectangle would be. The default mode for *ellipse()* is "CENTER", rather than "CORNER."




```
ellipseMode(CENTER);
ellipse(x,y,width,height);
```

Example:

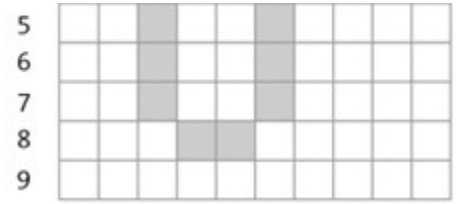
```
ellipseMode(CENTER);
ellipse(4,4,5,7);
```




```
ellipseMode(CORNER);
ellipse(x,y,width,height);
```

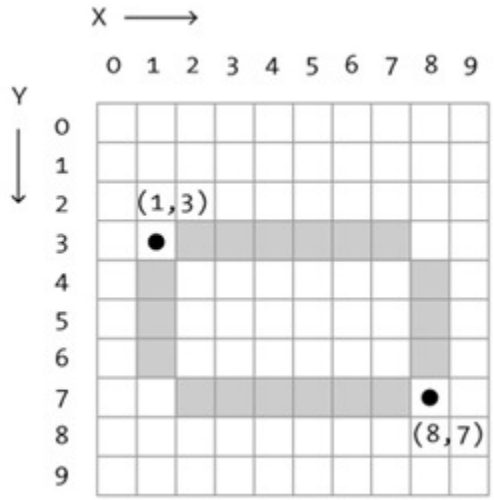
Example:

```
ellipseMode(CORNER);
ellipse(2,2,4,7);
```



Example:

```
ellipseMode(CORNER);
ellipse(2,2,4,7);
```



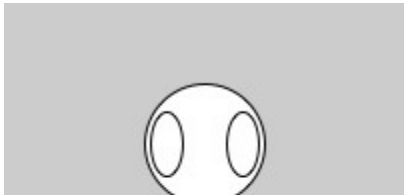

```
ellipseMode(CORNERS);
ellipse(x1,y1,x2,y2);
```

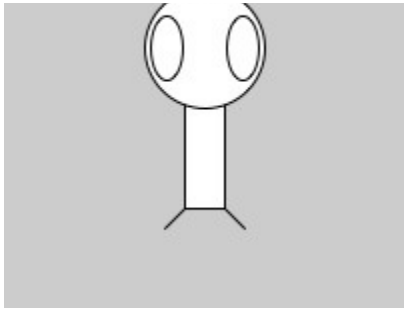
Example:

```
ellipseMode(CORNERS);
ellipse(1,3,8,7);
```

It is important to acknowledge that these ellipses do not look particularly circular. Processing has a built-in methodology for selecting which pixels should be used to create a circular shape. Zoomed in like this, we get a bunch of squares in a circle-like pattern, but zoomed out on a computer screen, we get a nice round ellipse. Processing also gives us the power to develop our own algorithms for coloring in individual pixels (in fact, we can already imagine how we might do this using "point" over and over again), but for now, we are content with allowing the "ellipse" statement to do the hard work. (For more about pixels, start with: the pixels reference page, though be warned this is a great deal more advanced than this tutorial.)

Now let's look at what some code with shapes in more realistic setting, with window dimensions of 200 by 200. Note the use of the size() function to specify the width and height of the window.





```
size(200,200);
rectMode(CENTER);
rect(100,100,20,100);
ellipse(100,70,60,60);
ellipse(81,70,16,32);
ellipse(119,70,16,32);
line(90,150,80,160);
line(110,150,120,160);
```

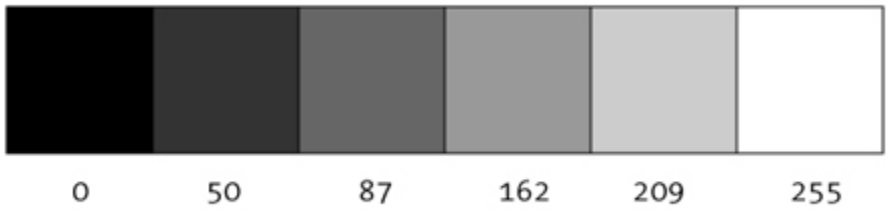
Colors

Color

This tutorial is for Processing version 1.1+. If you see any errors or have comments, please let us know. This tutorial is from the book, Learning Processing, by Daniel Shiffman, published by Morgan Kaufmann Publishers, Copyright © 2008 Elsevier Inc. All rights reserved.

Grayscale Color

In the digital world, when we want to talk about a color, precision is required. Saying "Hey, can you make that circle bluish-green?" will not do. Color, rather, is defined as a range of numbers. Let's start with the simplest case: black & white or grayscale. 0 means black, 255 means white. In between, every other number - 50, 87, 162, 209, and so on - is a shade of gray ranging from black to white.



Does 0-255 seem arbitary to you?
Color for a given shape needs to be stored in the computer's memory. This memory is just a long sequence of 0's and 1's (a whole bunch of on or off switches.) Each one of these switches is a bit, eight of them together is a byte. Imagine if we had eight bits (one byte) in sequence - how many ways can we configure these switches? The answer is (and doing a little research into binary numbers will prove this point) 256 possibilities, or a range of numbers between 0 and 255. We will use eight bit color for our grayscale range and 24 bit for full color (eight bits for each of the red, green, and blue color components).

and 255. We will use eight bit color for our grayscale range and 24 bit for full color (eight bits for each of the red, green, and blue color components).

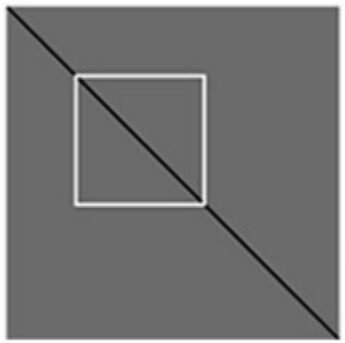
By adding the stroke() and fill() functions before something is drawn, we can set the color of any given shape. There is also the function background(), which sets a background color for the window. Here's an example.

```
background(255);    // Setting the background to white
stroke(0);          // Setting the outline (stroke) to black
fill(150);          // Setting the interior of a shape (fill) to grey
rect(50,50,75,100); // Drawing the rectangle
```

Stroke or fill can be eliminated with the functions: noStroke() and noFill(). Our instinct might be to say "stroke(0)" for no outline, however, it is important to remember that 0 is not "nothing", but rather denotes the color black. Also, remember not to eliminate both - with **noStroke()** and **noFill()**, nothing will appear!

In addition, if we draw two shapes, Processing will always use the most recently specified stroke and fill, reading the code from top to bottom.

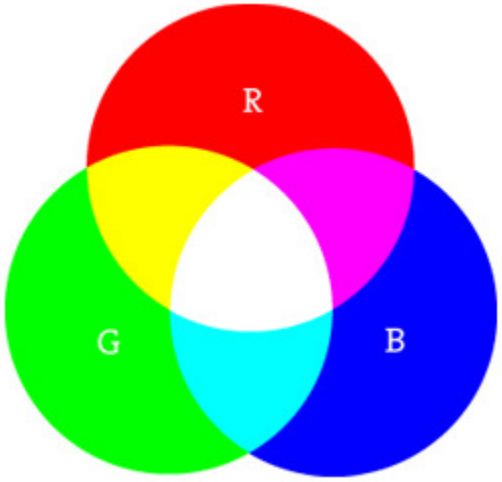
```
background(150);
stroke(0);
line(0,0,100,100);
stroke(255);
noFill();
rect(25,25,50,50);
```



RGB Color

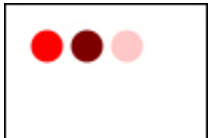
Remember finger painting? By mixing three "primary" colors, any color could be generated. Swirling all colors together resulted in a muddy brown. The more paint you added, the darker it got. Digital colors are also constructed by mixing

Remember finger painting? By mixing three "primary" colors, any color could be generated. Swirling all colors together resulted in a muddy brown. The more paint you added, the darker it got. Digital colors are also constructed by mixing three primary colors, but it works differently from paint. First, the primaries are diff erent: red, green, and blue (i.e., "RGB" color). And with color on the screen, you are mixing light, not paint, so the mixing rules are different as well.



- Red + Green = Yellow
- Red + Blue = Purple
- Green + Blue = Cyan (blue-green)
- Red + Green + Blue = White
- no colors = Black

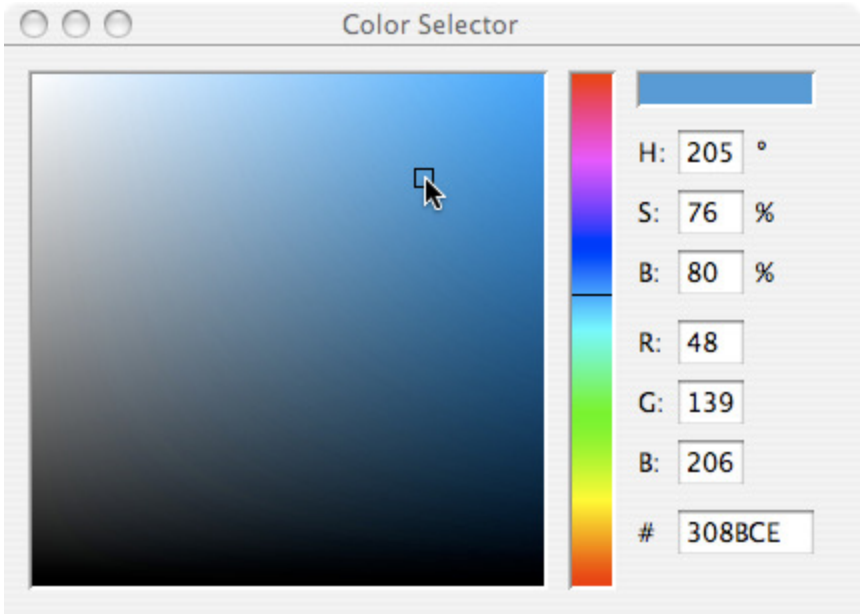
This assumes that the colors are all as bright as possible, but of course, you have a range of color available, so some red plus some green plus some blue equals gray, and a bit of red plus a bit of blue equals dark purple. While this may take some getting used to, the more you program and experiment with RGB color, the more it will become instinctive, much like swirling colors with your fi ngers. And of course you can't say "Mix some red with a bit of blue," you have to provide an exact amount. As with grayscale, the individual color elements are expressed as ranges from 0 (none of that color) to 255 (as much as possible), and they are listed in the order R, G, and B. You will get the hang of RGB color mixing through experimentation, but next we will cover some code using some common colors.



Example: RGB color

```
background(255);
noStroke();

// Bright red
fill(255,0,0);
ellipse(20,20,16,16);

// Dark red
fill(127,0,0);
ellipse(40,20,16,16);

// Pink (pale red)
fill(255,200,200);
ellipse(60,20,16,16);
```

Processing also has a color selector to aid in choosing colors. Access this via TOOLS (from the menu bar) → COLOR SELECTOR.



https://pythonhosted.org/ete2/reference/reference_svgcolors.html

Red colors

IndianRed	CD 5C 5C	205 92 92
LightCoral	F0 80 80	240 128 128
Salmon	FA 80 72	250 128 114
DarkSalmon	E9 96 7A	233 150 122
LightSalmon	FF A0 7A	255 160 122
Crimson	DC 14 3C	220 20 60
Red	FF 00 00	255 0 0
FireBrick	B2 22 22	178 34 34
DarkRed	8B 00 00	139 0 0

Pink colors

Pink	FF C0 CB	255 192 203
LightPink	FF B6 C1	255 182 193
HotPink	FF 69 B4	255 105 180
DeepPink	FF 14 93	255 20 147
MediumVioletRed	C7 15 85	199 21 133
PaleVioletRed	DB 70 93	219 112 147

Orange colors

LightSalmon	FF A0 7A	255 160 122
Coral	FF 7F 50	255 127 80
Tomato	FF 63 47	255 99 71
OrangeRed	FF 45 00	255 69 0
DarkOrange	FF 8C 00	255 140 0
Orange	FF A5 00	255 165 0

Yellow colors

Gold	FF D7 00	255 215 0
Yellow	FF FF 00	255 255 0
LightYellow	FF FF E0	255 255 224
LemonChiffon	FF FA CD	255 250 205
LightGoldenrodYellow	FA FA D2	250 250 210
PapayaWhip	FF EF D5	255 239 213
Moccasin	FF E4 B5	255 228 181
PeachPuff	FF DA B9	255 218 185
PaleGoldenrod	EE E8 AA	238 232 170
Khaki	F0 E6 8C	240 230 140
DarkKhaki	BD B7 6B	189 183 107

Purple colors

Lavender	E6 E6 FA	230 230 250
Thistle	D8 BF D8	216 191 216
Plum	DD A0 DD	221 160 221
Violet	EE 82 EE	238 130 238
Orchid	DA 70 D6	218 112 214
Fuchsia	FF 00 FF	255 0 255
Magenta	FF 00 FF	255 0 255
MediumOrchid	BA 55 D3	186 85 211
BlueViolet	8A 2B E2	138 43 226
DarkViolet	94 00 D3	148 0 211
DarkOrchid	99 32 CC	153 50 204
DarkMagenta	8B 00 8B	139 0 139
Purple	80 00 80	128 0 128
Indigo	4B 00 82	75 0 130
SlateBlue	6A 5A CD	106 90 205
DarkSlateBlue	48 3D 8B	72 61 139
MediumSlateBlue	7B 68 EE	123 104 238

Green colors

GreenYellow	AD FF 2F	173 255 47
Chartreuse	7F FF 00	127 255 0
LawnGreen	7C FC 00	124 252 0
Lime	00 FF 00	0 255 0
LimeGreen	32 CD 32	50 205 50
PaleGreen	98 FB 98	152 251 152
LightGreen	90 EE 90	144 238 144
MediumSpringGreen	00 FA 9A	0 250 154
SpringGreen	00 FF 7F	0 255 127
MediumSeaGreen	3C B3 71	60 179 113
SeaGreen	2E 8B 57	46 139 87
ForestGreen	22 8B 22	34 139 34
Green	00 80 00	0 128 0
DarkGreen	00 64 00	0 100 0
YellowGreen	9A CD 32	154 205 50
OliveDrab	6B 8E 23	107 142 35
Olive	80 80 00	128 128 0
DarkOliveGreen	55 6B 2F	85 107 47
MediumAquamarine	66 CD AA	102 205 170
DarkSeaGreen	8F BC 8F	143 188 143
LightSeaGreen	20 B2 AA	32 178 170
DarkCyan	00 8B 8B	0 139 139
Teal	00 80 80	0 128 128

Blue/Cyan colors

Aqua	00 FF FF	0 255 255
Cyan	00 FF FF	0 255 255
LightCyan	E0 FF FF	224 255 255
PaleTurquoise	AF EE EE	175 238 238
Aquamarine	7F FF D4	127 255 212
Turquoise	40 E0 D0	64 224 208
MediumTurquoise	48 D1 CC	72 209 204
DarkTurquoise	00 CE D1	0 206 209
CadetBlue	5F 9E A0	95 158 160
SteelBlue	46 82 B4	70 130 180
LightSteelBlue	B0 C4 DE	176 196 222
PowderBlue	B0 E0 E6	176 224 230
LightBlue	AD D8 E6	173 216 230
SkyBlue	87 CE EB	135 206 235
LightSkyBlue	87 CE FA	135 206 250
DeepSkyBlue	00 BF FF	0 191 255
DodgerBlue	1E 90 FF	30 144 255
CornflowerBlue	64 95 ED	100 149 237
MediumSlateBlue	7B 68 EE	123 104 238
RoyalBlue	41 69 E1	65 105 225
MediumBlue	00 00 CD	0 0 205
DarkBlue	00 00 8B	0 0 139
Navy	00 00 80	0 0 128
MidnightBlue	19 19 70	25 25 112

Brown colors

Cornsilk	FF F8 DC	255 248 220
BlanchedAlmond	FF EB CD	255 235 205
Bisque	FF E4 C4	255 228 196
NavajoWhite	FF DE AD	255 222 173
Wheat	F5 DE B3	245 222 179
BurlyWood	DE B8 87	222 184 135
Tan	D2 B4 8C	210 180 140
RosyBrown	BC 8F 8F	188 143 143
SandyBrown	F4 A4 60	244 164 96
Goldenrod	DA A5 20	218 165 32
DarkGoldenrod	B8 86 0B	184 134 11
Peru	CD 85 3F	205 133 63
Chocolate	D2 69 1E	210 105 30
SaddleBrown	8B 45 13	139 69 19
Sienna	A0 52 2D	160 82 45
Brown	A5 2A 2A	165 42 42
Maroon	80 00 00	128 0 0

White colors

White	FF FF FF	255 255 255
Snow	FF FA FA	255 250 250
Honeydew	F0 FF F0	240 255 240
MintCream	F5 FF FA	245 255 250
Azure	F0 FF FF	240 255 255
AliceBlue	F0 F8 FF	240 248 255
GhostWhite	F8 F8 FF	248 248 255
WhiteSmoke	F5 F5 F5	245 245 245
Seashell	FF F5 EE	255 245 238
Beige	F5 F5 DC	245 245 220
OldLace	FD F5 E6	253 245 230
FloralWhite	FF FA F0	255 250 240
Ivory	FF FF F0	255 255 240
AntiqueWhite	FA EB D7	250 235 215
Linen	FA F0 E6	250 240 230
LavenderBlush	FF F0 F5	255 240 245
MistyRose	FF E4 E1	255 228 225

Gray colors

Gainsboro	DC DC DC	220 220 220
LightGrey	D3 D3 D3	211 211 211
Silver	C0 C0 C0	192 192 192
DarkGray	A9 A9 A9	169 169 169
Gray	80 80 80	128 128 128
DimGray	69 69 69	105 105 105
LightSlateGray	77 88 99	119 136 153
SlateGray	70 80 90	112 128 144
Black	00 00 00	0 0 0

Python Comments

How to use comments in Python

Comments

Comments in Python are used to explain what the code does.

Python Comments

Python has two ways to annotate Python code.

One is by using comments to indicate what some part of the code does.

Single-line comments begin with the hash character ("#") and are terminated by the end of line.

Python is ignoring all text that comes after the # to the end of the line, they are not part of the command.

Comments spanning more than one line are achieved by inserting a multi-line string (with """ as the delimiter one each end) that is not used in assignment or otherwise evaluated, but sits in between other statements.

They are meant as documentation for anyone reading the code.

Example

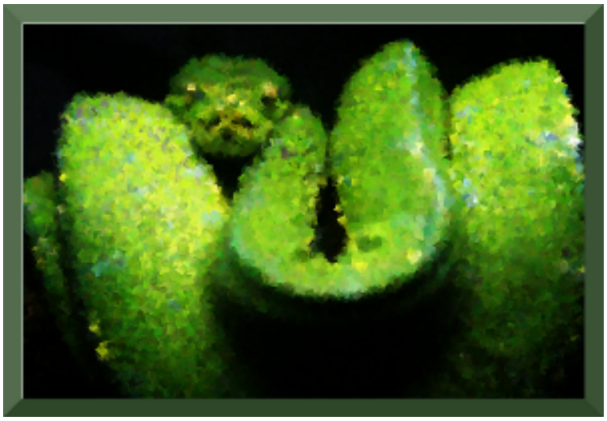
Let's show this by using an example

```
#this is a comment in Python

Print("Hello World") #This is also a comment in Python

""" This is an example of a multiline
comment that spans multiple lines
...
"""
```

The Canvas Object

CANVAS WIDGETS

INTRODUCTION



The Canvas widget supplies graphics facilities for Tkinter. Among these graphical objects are lines, circles, images, and even other widgets. With this widget it's possible to draw graphs and plots, create graphics editors, and implement various kinds of custom widgets.

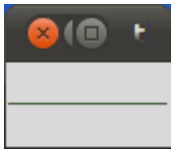
We demonstrate in our first example, how to draw a line. The method create_line(coords, options) is used to draw a straight line. The coordinates "coords" are given as four integer numbers: x_1, y_1, x_2, y_2 This means that the line goes from the point (x_1, y_1) to the point (x_2, y_2) After these coordinates follows a comma separated list of additional parameters, which may be empty. We set for example the colour of the line to the special green of our website: fill="#476042"

We kept the first example intentionally very simple. We create a canvas and draw a straight horizontal line into this canvas. This line vertically cuts the canvas into two areas.

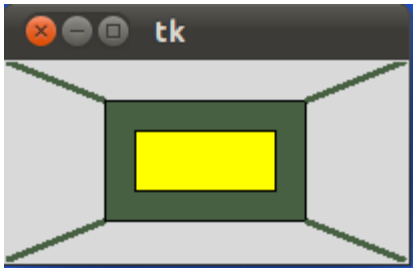
The casting to an integer value in the assignment "y = int(canvas_height / 2)" is superfluous, because create_line can work with float values as well. They are automatically turned into integer values. In the following you can see the code of our first simple script:

```
from tkinter import *
master = Tk()

canvas_width = 80
canvas_height = 40
w = Canvas(master,
           width=canvas_width,
           height=canvas_height)
w.pack()

y = int(canvas_height / 2)
w.create_line(0, y, canvas_width, y, fill="#476042")


mainloop()
```

If we start this program, using Python 3, we get the following window:



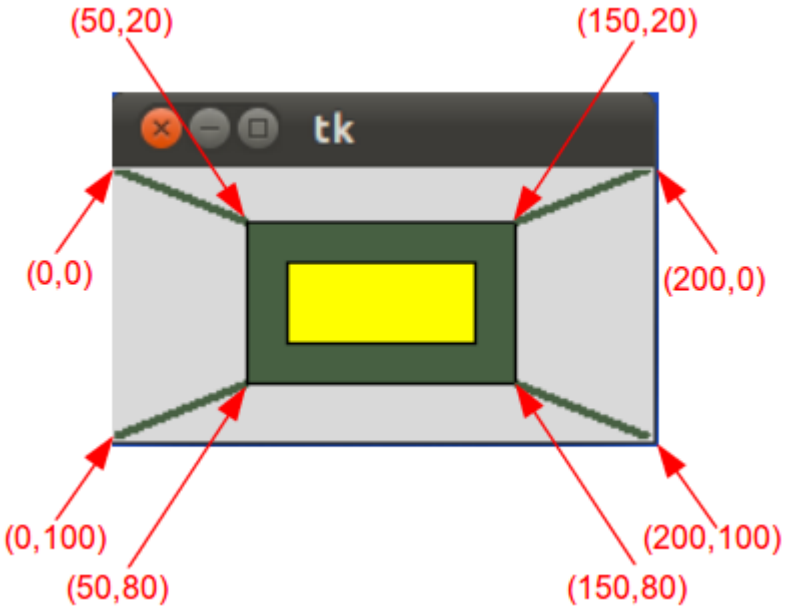
For creating rectangles we have the method create_rectangle(coords, options). Coords is again defined by two points, but this time the first one is the top left point and the bottom right point of the rectangle.



The window, you see above, is created by the following Python tkinter code:

```
from tkinter import *

master = Tk()

w = Canvas(master, width=200, height=100)
w.pack()

w.create_rectangle(50, 20, 150, 80, fill="#476042")
w.create_rectangle(65, 35, 135, 65, fill="yellow")
w.create_line(0, 0, 50, 20, fill="#476042", width=3)
w.create_line(0, 100, 50, 80, fill="#476042", width=3)
w.create_line(150,20, 200, 0, fill="#476042", width=3)
w.create_line(150, 80, 200, 100, fill="#476042", width=3)

mainloop()
```

The following image with the coordinates will simplify the understanding of application of create_lines and create_rectangle in our previous example.



TEXT ON CANVAS

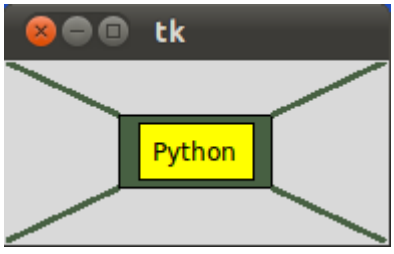
We demonstrate now how to print text on a canvas. We will extend and modify the previous example for this purpose. The method create_text() can be applied to a canvas object to write text on it. The first two parameters are the x and the y positions of the text object. By default, the text is centred on this position. You can override this with the anchor option. For example, if the coordinate should be the upper left corner, set the anchor to NW. With the keyword parameter text, we can define the actual text to be displayed on the canvas.

```
from tkinter import *

canvas_width = 200
canvas_height = 100

colours = ("#476042", "yellow")
box=[]

for ratio in ( 0.2, 0.35 ):
```

```
    box.append( (canvas_width * ratio,
                 canvas_height * ratio,
                 canvas_width * (1 - ratio),
                 canvas_height * (1 - ratio) ) )

master = Tk()

w = Canvas(master,
           width=canvas_width,
           height=canvas_height)
w.pack()

for i in range(2):
    w.create_rectangle(box[i][0], box[i][1],box[i][2],box[i][3], fill=colours[i])

w.create_line(0, 0,                 # origin of canvas
              box[0][0], box[0][1], # coordinates of left upper corner of the box[0]
              fill=colours[0],
              width=3)
w.create_line(0, canvas_height,     # lower left corner of canvas
              box[0][0], box[0][3], # lower left corner of box[0]
              fill=colours[0],
              width=3)
w.create_line(box[0][2],box[0][1],  # right upper corner of box[0]
              canvas_width, 0,      # right upper corner of canvas
              fill=colours[0],
              width=3)
w.create_line(box[0][2], box[0][3], # lower right corner pf box[0]
              canvas_width, canvas_height, # lower right corner of canvas
              fill=colours[0], width=3)

w.create_text(canvas_width / 2,
              canvas_height / 2,
              text="Python")
mainloop()
```

Though the code of our example program is changed drastically, the graphical result looks still the same except for the text "Python":

You can understand the benefit of our code changes, if you change for example the height of the canvas to 190 and the width to 90 and modify the ratio for the first box to 0.3. Image doing this in the code of our first example. It would be a lot tougher. The result looks like this:



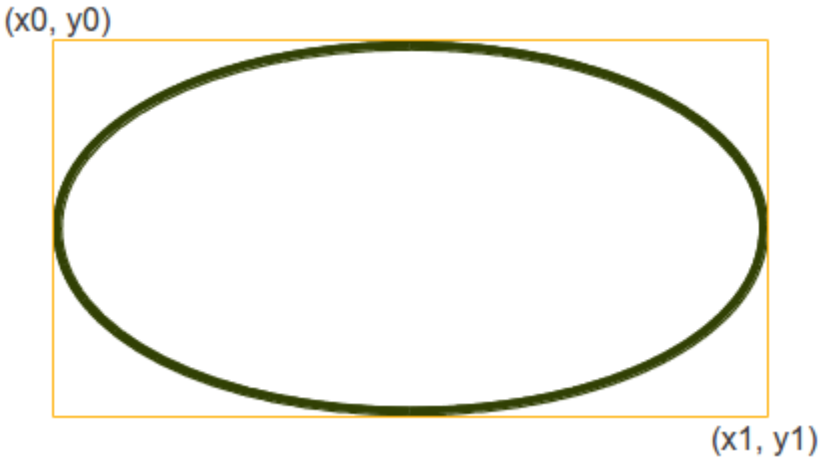
OVAL OBJECTS

An oval (or an ovoid) is any curve resembling an egg (ovum means egg in Latin). It resembles an ellipse, but it is not an ellipse. The term "oval" is not well-defined. Many different curves are called ovals, but they all have in common:

- They are differentiable, simple (not self-intersecting), convex, closed, plane curves
- They are very similar in shape to ellipses
- There is at least one axis of symmetry

The word oval stems from Latin ovum meaning "egg" and that's what it is: A figure which resembles the form of an egg. An oval is constructed from two pairs of

arcs, with two different radii A circle is a special case of an oval.



We can create an oval on a canvas c with the following method:

```
id = C.create_oval ( x0, y0, x1, y1, option, ... )
```

This method returns the object ID of the new oval object on the canvas C.

The following script draws a circle around the point (75,75) with the radius 25:

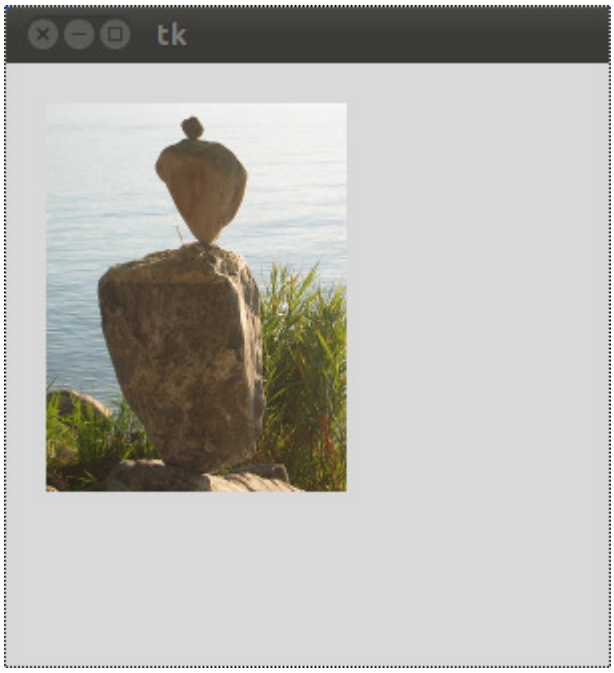
```
from tkinter import *

canvas_width = 190
canvas_height =150

master = Tk()

w = Canvas(master,
           width=canvas_width,
           height=canvas_height)
w.pack()

w.create_oval(50,50,100,100)

mainloop()
```

THE CANVAS IMAGE ITEM

The Canvas method create_image(x0,y0, options ...) is used to draw an image on a canvas. create_image doesn't accept an image directly. It uses an object which is created by the PhotoImage() method. The PhotoImage class can only read GIF and PGM/PPM images from files

```
from tkinter import *

canvas_width = 300
canvas_height =300

master = Tk()

canvas = Canvas(master,
           width=canvas_width,
           height=canvas_height)
canvas.pack()

img = PhotoImage(file="rocks.ppm")
canvas.create_image(20,20, anchor=NW, image=img)

mainloop()
```

The window created by the previous Python script looks like this:



PYTHON TKINTER CANVAS

The Canvas is a rectangular area intended for drawing pictures or other complex layouts. You can place graphics, text, widgets or frames on a Canvas.

Syntax:

Here is the simple syntax to create this widget:

```
w = Canvas ( master, option=value, ... )
```

Parameters:

- **master:** This represents the parent window.
- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
bd	Border width in pixels. Default is 2.
bg	Normal background color.
confine	If true (the default), the canvas cannot be scrolled outside of the scrollregion.
cursor	Cursor used in the canvas like *arrow, circle, dot etc.*
height	Size of the canvas in the Y dimension.
highlightcolor	Color shown in the focus highlight.
relief	Relief specifies the type of the border. Some of the values are SUNKEN, RAISED, GROOVE, and RIDGE.
scrollregion	A tuple (w, n, e, s) that defines over how large an area the canvas can be scrolled, where w is the left side, n the top, e the right side, and s the bottom.
width	Size of the canvas in the X dimension.
xscrollincrement	If you set this option to some positive dimension, the canvas can be positioned only on multiples of that distance, and the value will be used for scrolling by scrolling units, such as when the user clicks on the arrows at the ends of a scrollbar.
xscrollcommand	If the canvas is scrollable, this attribute should be the .set() method of the horizontal scrollbar.
yscrollincrement	Works like xscrollincrement, but governs vertical movement.
yscrollcommand	If the canvas is scrollable, this attribute should be the .set() method of the vertical scrollbar.

The Canvas widget can support the following standard items:

arc . Creates an arc item, which can be a chord, a pieslice or a simple arc.

```
coord = 10, 50, 240, 210
arc = canvas.create_arc(coord, start=0, extent=150, fill="blue")
```

image . Creates an image item, which can be an instance of either the BitmapImage or the PhotoImage classes.

```
filename = PhotoImage(file = "sunshine.gif")
image = canvas.create_image(50, 50, anchor=NE, image=filename)
```

line . Creates a line item.

```
line = canvas.create_line(x0, y0, x1, y1, ..., xn, yn, options)
```

oval . Creates a circle or an ellipse at the given coordinates. It takes two pairs of coordinates; the top left and bottom right corners of the bounding rectangle for the oval.

```
oval = canvas.create_oval(x0, y0, x1, y1, options)
```

polygon . Creates a polygon item that must have at least three vertices.

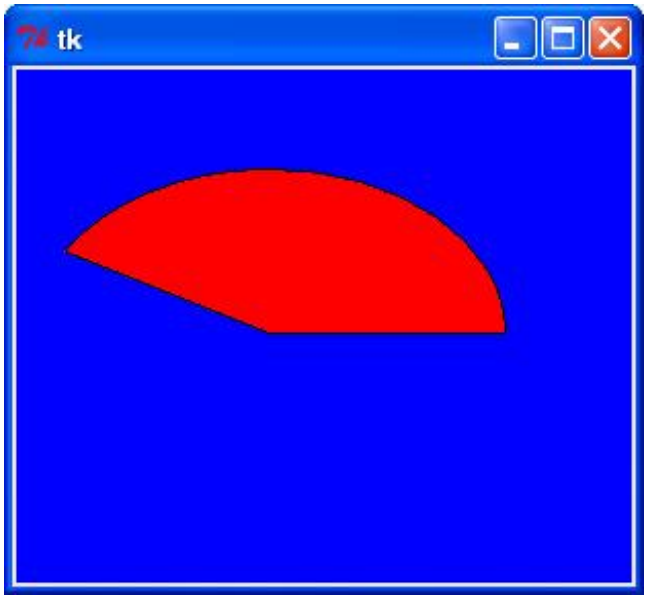
```
oval = canvas.create_polygon(x0, y0, x1, y1,...xn, yn, options)
```

Example:

Try the following example yourself:

```
import Tkinter
import tkMessageBox

top = Tkinter.Tk()

C = Tkinter.Canvas(top, bg="blue", height=250, width=300)

coord = 10, 50, 240, 210
arc = C.create_arc(coord, start=0, extent=150, fill="red")

C.pack()
top.mainloop()
```

When the above code is executed, it produces the following result:



Console Output

PRINT

INTRODUCTION

In principle, every computer program has to communicate with the environment or the "outside world". To this purpose nearly every programming language has special I/O functionalities, i.e. input/output. This ensures the interacttion or communication with other components e.g. a data base or a user. Input often comes - as we have already seen - from the keyboard and the corresponding Python command or better the corresponding Python function for reading from the standard input is input().

We have also seen in previous examples of our tutorial, that we can write into the standard output by using print. In this chapter of our tutorial we want to have a detailed look at the print function. As some might have skipped over it, we want to emphasize that we wrote "print function" and not "print statement". You can easily find out how crucial this difference is, if you take an arbitrary Python program written in version 2.x and if you try to let it run with a Python3 interpreter. In most cases you will receive error messages. One of the most frequently occurring errors will be related to print, because most programs contain prints. We can generate the most typical error in the interactive Python shell:

```
$ python3
Python 3.2.3 (default, Apr 10 2013, 05:03:36)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license"
for more information.
>>> print 42
  File "", line 1
    print 42
          ^
SyntaxError: invalid syntax
>>>
```

This is a familiar error message for most of us: We have forgotten the parentheses. "print" is - as we have already mentioned - a function in version 3.x. Like any other function print expects its arguments to be surrounded by parentheses. So parenthesis are an easy remedy for this error:

```
>>> print(42)
42
>>>
```

But this is not the only difference to the old print. The output behaviour has changed as well:

PRINT FUNCTION

The arguments of the print function are the following ones:

```
print(value1, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

The print function can print an arbitrary number of values ("value1, value2, ..."), which are seperated by commas. die durch Komma getrennt sind. These values are seperated by blanks. In the following example we can see two print calls. We are printing two values in both cases, i.e. a string and a float number:

```
>>> print("a = ", a)
a =  3.564
>>> print("a = \n", a)
a =
 3.564
>>>
```

We can learn from the second print of the example, that a blank between two values, i.e. "a = \textbackslash n" and "3.564", is always printed, even if the output is continued in the following line. This is different to Python 2, as there will be no blank printed, if a new line has been started. It's possible to redefine the sepeerator between values by assigning an arbitrary string to the keyword parameter "sep", e.e. an empty string or a smiley:

```
>>> print("a","b")
a b
>>> print("a","b",sep="")
ab
>>> print(192,168,178,42,sep=".")
192.168.178.42
>>> print("a","b",sep=":-)")
a:-)b
>>>
```

A print call is ended by a newline, as we can see in the following usage:

```
>>> for i in range(4):
...     print(i)
...
0
1
2
3
>>>
```

To change this behaviour, we can assign an arbitrary string to the keyword parameter "end". This string will be used for ending the output of the values of a print call:

```
>>> for i in range(4):
...     print(i, end=" ")
...
0 1 2 3 >>>
>>> for i in range(4):
...     print(i, end=" :-) ")
...
0 :-) 1 :-) 2 :-) 3 :-) >>>
```

The output of the print function is send to the standard output stream (sys.stdout) by default. By redefining the keyword parameter "file" we can send the output into a different stream e.g. sys.stderr or a file:

```
>>> fh = open("data.txt","w")
>>> print("42 is the answer, but what is the question?", file=fh)
>>> fh.close()
>>>
```

We can see, that we don't get any output in the interactive shell. The output is sent to the file "data.txt". It's also possible to redirect the output to the standard error channel this way:

```
>>> import sys
>>> # output into sys.stderr:
...
>>> print("Error: 42", file=sys.stderr)
Error: 42
```