



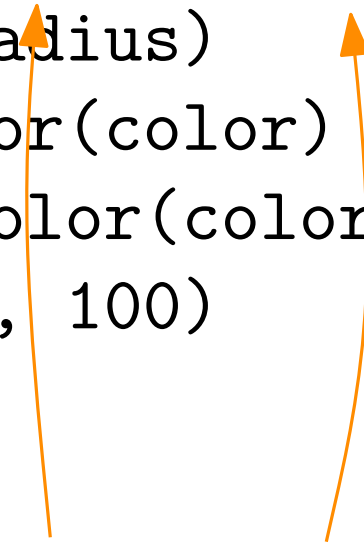
We have learnt about **parameters** and **function arguments**:

```
def create_sun(radius, color):  
    sun = Circle(radius)  
    sun.setFillColor(color)  
    sun.setBorderColor(color)  
    sun.moveTo(100, 100)  
    return sun  
  
sun = create_sun(30, "yellow")
```



We have learnt about **parameters** and function **arguments**:

```
def create_sun(radius, color):  
    sun = Circle(radius)  
    sun.setFillColor(color)  
    sun.setBorderColor(color)  
    sun.moveTo(100, 100)  
    return sun  
  
sun = create_sun(30, "yellow")
```





We have learnt about **parameters** and function **arguments**:

```
def create_sun(radius, color):  
    sun = Circle(radius)  
    sun.setFillColor(color)  
    sun.setBorderColor(color)  
    sun.moveTo(100, 100)  
    return sun
```

```
sun = create_sun(30, "yellow")
```

Arguments are mapped to parameters one-by-one, left-to-right.



We can provide default parameters:

```
def create_sun(radius = 30, color = "yellow"):
    # as before
```



We can provide default parameters:

```
def create_sun(radius = 30, color = "yellow"):
    # as before
```

Now we can call it like this:

```
sun = create_sun()
star = create_sun(2)
moon = create_sun(28, "silver")
```



We can provide default parameters:

```
def create_sun(radius = 30, color = "yellow"):
    # as before
```

Now we can call it like this:

```
sun = create_sun()
star = create_sun(2)
moon = create_sun(28, "silver")
```

But not like this:

```
moon = create_sun("silver")
```



We can provide default parameters:

```
def create_sun(radius = 30, color = "yellow"):
    # as before
```

Now we can call it like this:

```
sun = create_sun()
star = create_sun(2)
moon = create_sun(28, "silver")
```

But not like this:

```
moon = create_sun("silver")
```



Default parameters have to follow normal parameters:

```
def avg(data, start = 0, end = None):  
    if not end:  
        end = len(data)  
    return sum(data[start:end]) / float(end-start)
```




Default parameters have to follow normal parameters:

```
def avg(data, start = 0, end = None):  
    if not end:  
        end = len(data)  
    return sum(data[start:end]) / float(end-start)
```

```
>>> d = [ 1, 2, 3, 4, 5 ]
```

```
>>> avg(d)
```

```
3.0
```

```
>>> avg(d, 2)
```

```
4.0
```

```
>>> avg(d, 1, 4)
```

```
3.0
```



We can include the name of the parameter in the function call to make the code clearer. Then the order of arguments does not matter:

```
moon = create_sun(color = "silver")
```

```
moon = create_sun(color = "silver", radius = 28)
```



We can include the name of the parameter in the function call to make the code clearer. Then the order of arguments does not matter:

```
moon = create_sun(color = "silver")
```

```
moon = create_sun(color = "silver", radius = 28)
```

```
>>> avg(d, end=3)
```

```
2.0
```

```
>>> avg(data=d, end=3)
```

```
2.0
```

```
>>> avg(end=3, data=d)
```

```
2.0
```

```
>>> avg(end=3, d)
```

```
SyntaxError: non-keyword arg after keyword arg
```



We often want to produce nicely formatted output:

```
print "Max between " + str(x0) + " and " +  
      str(x1) + " is " + str(val)
```



We often want to produce nicely formatted output:

```
print "Max between " + str(x0) + " and " +  
      str(x1) + " is " + str(val)
```

The string formatting operator **%** makes this much easier:

```
print "Max between %d and %d is %g" % (x0, x1, val)
```



We often want to produce nicely formatted output:

```
print "Max between " + str(x0) + " and " +  
      str(x1) + " is " + str(val)
```

The string formatting operator **%** makes this much easier:

```
print "Max between %d and %d is %g" % (x0, x1, val)
```

Formatting operator:

```
format_string % (arg0, arg1, .... )
```



We often want to produce nicely formatted output:

```
print "Max between " + str(x0) + " and " +  
      str(x1) + " is " + str(val)
```

The string formatting operator `%` makes this much easier:

```
print "Max between %d and %d is %g" % (x0, x1, val)
```

Formatting operator:

```
format_string % (arg0, arg1, .... )
```

Tuple has one element for each **place holder** in the **format_string**. Place holders are:

- `%d` for integers in decimal,
- `%g` for `float`,
- `%.2f` for `float` with fixed precision (2 digits after period),
- `%s` for anything (like `str(x)`).



If there is only one place holder, tuple is not necessary:

```
print "Maximum is %g" % val
```




If there is only one place holder, tuple is not necessary:

```
print "Maximum is %g" % val
```

We can align tables by using field width:

```
print "%3d ~ %3d : %10g" % (x0, x1, val)
```



If there is only one place holder, tuple is not necessary:

```
print "Maximum is %g" % val
```

We can align tables by using field width:

```
print "%3d ~ %3d : %10g" % (x0, x1, val)
```

A value can be left-aligned in its field:

```
print "%3d ~ %-3d : %-12g" % (x0, x1, val)
```



Strings are sequences:

```
def is_palindrome(s):  
    for i in range(len(s) / 2):  
        if s[i] != s[len(s) - i - 1]:  
            return False  
    return True
```



Strings are sequences:

```
def is_palindrome(s):  
    for i in range(len(s) / 2):  
        if s[i] != s[len(s) - i - 1]:  
            return False  
    return True
```

Strings are **immutable**.



Strings are sequences:

```
def is_palindrome(s):  
    for i in range(len(s) / 2):  
        if s[i] != s[len(s) - i - 1]:  
            return False  
    return True
```

Strings are **immutable**.

The **in** operator for strings:

```
>>> "abc" in "01234abcdefg"  
True  
>>> "abce" in "01234abcdefg"  
False
```



Strings are sequences:

```
def is_palindrome(s):  
    for i in range(len(s) / 2):  
        if s[i] != s[len(s) - i - 1]:  
            return False  
    return True
```

Strings are **immutable**.

The **in** operator for strings:

```
>>> "abc" in "01234abcdefg"  
True  
>>> "abce" in "01234abcdefg"  
False
```

Different from the **in** operator for lists and tuples, which tests whether something is equal to an element of the list or tuple.



String objects have many useful methods:

- `upper()`, `lower()`, and `capitalize()`
- `isalpha()` and `isdigit()`
- `startswith(prefix)` and `endswith(suffix)`
- `find(str)`, `find(str, start)`, and `find(str, start, end)`
(return -1 if `str` is not in the string)
- `replace(str1, str2)`
- `rstrip()`, `lstrip()` and `strip()` to remove white space on the right, left, or both ends.

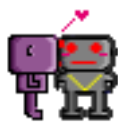


String objects have many useful methods:

- `upper()`, `lower()`, and `capitalize()`
- `isalpha()` and `isdigit()`
- `startswith(prefix)` and `endswith(suffix)`
- `find(str)`, `find(str, start)`, and `find(str, start, end)`
(return -1 if `str` is not in the string)
- `replace(str1, str2)`
- `rstrip()`, `lstrip()` and `strip()` to remove white space on the right, left, or both ends.

String methods for converting between lists and strings:

- `split()` splits with white space as separator
- `split(sep)` splits with given separator `sep`
- `join(l)` concatenates strings from a list `l`



You all know the mastermind game:





You all know the mastermind game:



We implement a simple text version: colored pegs are the six letters A - F. The secret has four letters, all distinct.



You all know the mastermind game:



We implement a simple text version: colored pegs are the six letters A - F. The secret has four letters, all distinct.

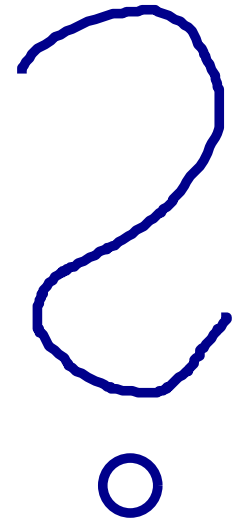
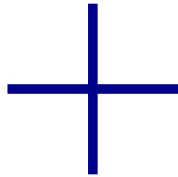
```
1: ABCD : 1 positions, 2 letters
2: ABDF : 0 positions, 3 letters
3: CAFB : 1 positions, 1 letters
4: EBAD : 2 positions, 2 letters
5: BEAD : 1 positions, 3 letters
```

Enter your guess> EABD

My secret was EABD, you guessed in 6 guesses!

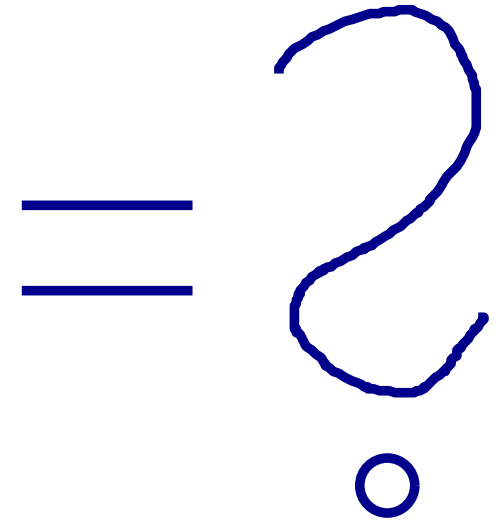
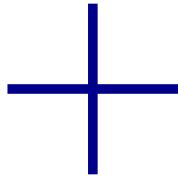


Let's put the KAIST statue on a nice background:





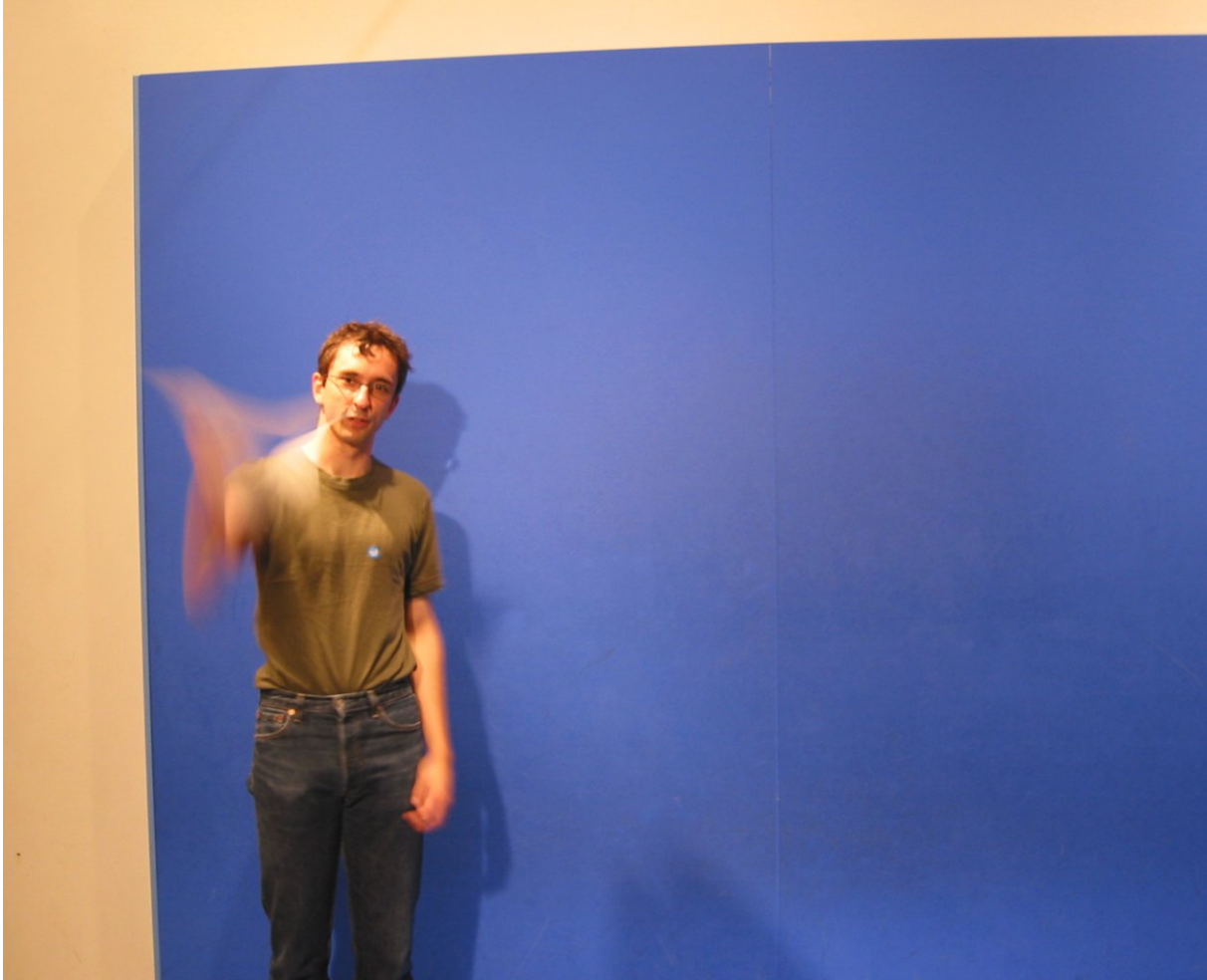
Let's put the KAIST statue on a nice background:



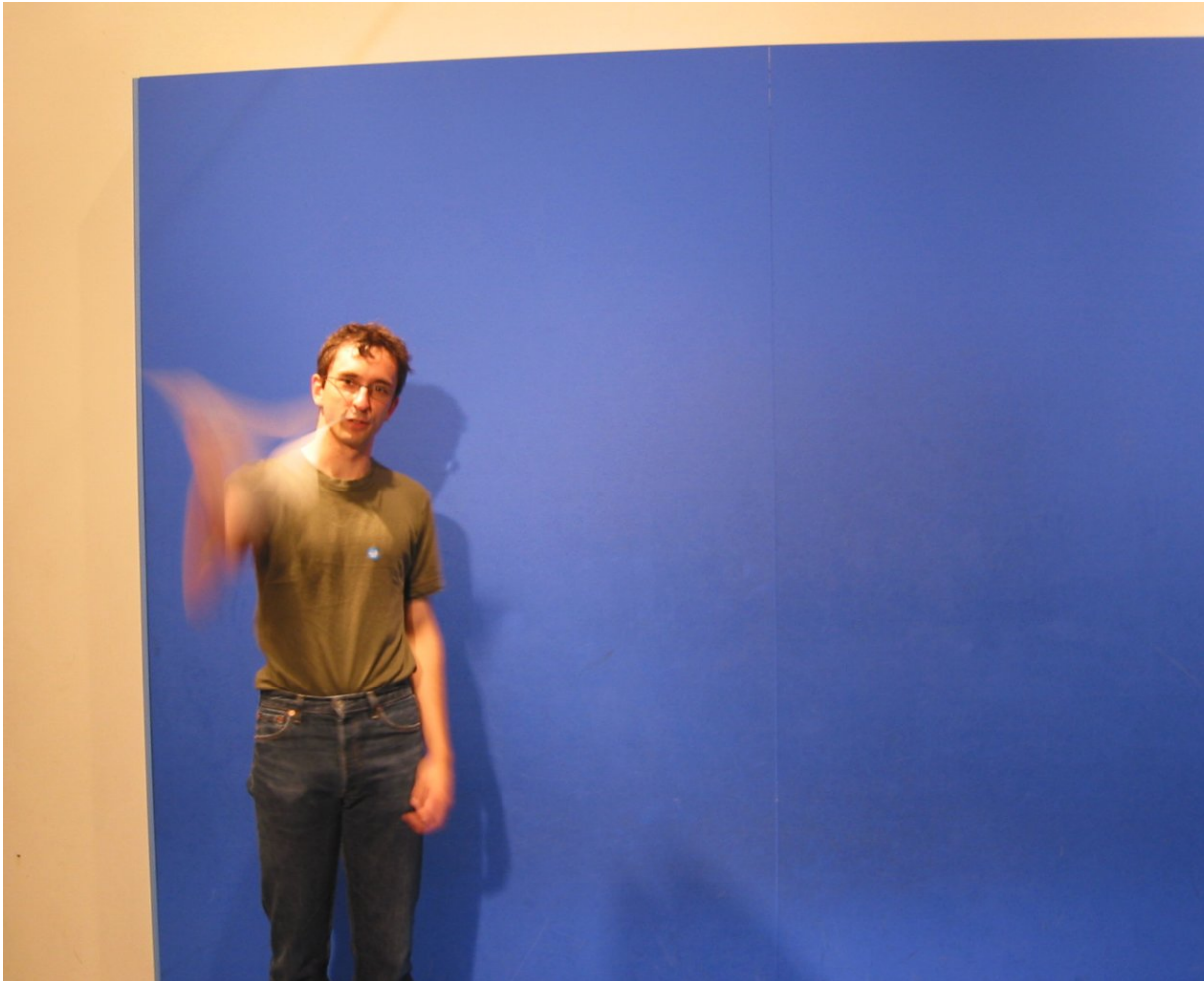
```
def paste(canvas, img, x1, y1):  
    w, h = img.size()  
    for y in range(h):  
        for x in range(w):  
            canvas.set(x1 + x, y1 + y, img.get(x, y))
```




Chromakey is a technique to overlay one scene on top of another one. It is commonly used for weather maps.



Chromakey is a technique to overlay one scene on top of another one. It is commonly used for weather maps.





Actually, the background is not exactly blue - just blueish.
We need a function to decide how similar two colors are:

```
def dist(c1, c2):  
    r1, g1, b1 = c1  
    r2, g2, b2 = c2  
    return math.sqrt((r1-r2)**2 + (g1-g2)**2 +  
                      (b1-b2)**2)
```




Actually, the background is not exactly blue - just blueish.
We need a function to decide how similar two colors are:

```
def dist(c1, c2):  
    r1, g1, b1 = c1  
    r2, g2, b2 = c2  
    return math.sqrt((r1-r2)**2 + (g1-g2)**2 +  
                      (b1-b2)**2)
```

This is just the Euclidean distance in \mathbb{R}^3 .



```
def chroma(img, key, threshold):  
    w, h = img.size()  
    for y in range(h):  
        for x in range(w):  
            p = img.get(x, y)  
            if dist(p, key) < threshold:  
                img.set(x, y, Color.yellow)
```





Now all we need is a paste function that skips the color-coded background:

```
def chroma_paste(canvas, img, x1, y1, key):  
    w, h = img.size()  
    for y in range(h):  
        for x in range(w):  
            p = img.get(x, y)  
            if p != key:  
                canvas.set(x1 + x, y1 + y, p)
```





Humans cannot perceive a small change in light intensity or color value. We can use this to hide information inside images.



Humans cannot perceive a small change in light intensity or color value. We can use this to hide information inside images.

Here is an algorithm to hide a black/white image `secret` in an image `img`:

- For all pixels (r, g, b) of `img`, if `r` is odd then subtract one from `r`;
- For each black pixel of `secret`, add one to the red value of the same pixel in `img`.



Humans cannot perceive a small change in light intensity or color value. We can use this to hide information inside images.

Here is an algorithm to hide a black/white image `secret` in an image `img`:

- For all pixels (r, g, b) of `img`, if `r` is odd then subtract one from `r`;
- For each black pixel of `secret`, add one to the red value of the same pixel in `img`.

To decode the secret, we look at all pixels (r, g, b) of the image, and turn it black if `r` is odd, and white otherwise.