



Introduction to Python

What exactly is Python?

Python is one of the most popular programming languages among software developers. It is a **general-purpose language**, therefore it can be used for a variety of tasks, ranging from machine learning and AI to web development.

Its design philosophy encompasses **code readability** through the implementation of straightforward and intuitive functions, code indentation and many more.

Getting used to the syntax

syntax = code structure (roughly equivalent to orthography rules of a written language)

If you're familiar with other programming languages like C++ or Java, you've seen **brackets** (`{...}`) a lot. They're used to mark the scope (bounds) of a function or a control structure. Python comes in with a difference: it replaces brackets with a **5-space indent (tab)**.

Here's a C++ vs Python code comparison:

```
int sum(int a, int b)
{
    int s = a + b;
    return s;
}
```

C++

```
def sum(a, b):
    s = a+b
    return s
```

Python

Getting used to the syntax

Furthermore, programming languages seem to love the **null statement** (**semicolon ;**) used at the end of another statement (which actually tells the computer to do something). Again, Python disposed of this feature (as you can observe in the previously given example).

Python comes up with a new syntax element, the **colon (:)**, which marks the beginning of the function body/the control structure instructions. Again, this can be seen on the previous slide.

Everybody has a different way of thinking. This is why **comments** are so useful. These are the parts of the code which are neither interpreted nor executed and they usually give information about how the code works. The **hash (#)** marks the beginning of a comment.

```
def sum(a, b):  
    s = a+b # creates a variable to store the sum  
    return s  
  
print(sum(100, 1000)) # has no indentation = not inside the function  
# comments span a single line
```

Data types

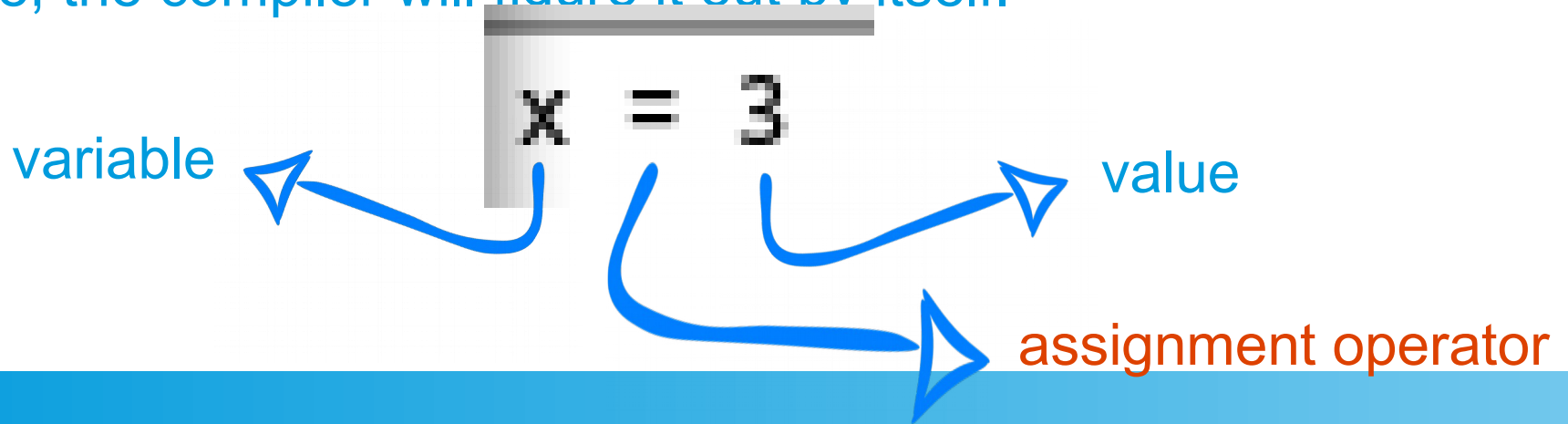
In order to have a functioning program, we need **data**. This data can be anything: numerical values, text, various structures and so on. This variety of forms in which data appears has led to the emergence of the primordial concept of **data types**. These help distinguish various forms of data and associate suitable **functions/operations** to them. For example, the `+` operator acts different with numbers than it does with text.

In programming (Python makes no exception), data is stored in **variables**. Think of them as keywords used to access certain values.

Number-related types

The most basic (and also the most important!) data types are numerical. These come in 3 different forms: the **integer (int)** and the **floating-point decimal (float)**.

In Python, there is no need to specify the type when declaring a variable, the compiler will figure it out by itself.



Number-related types

Question time!

1. Create a variable `x` containing any integer value.
2. Create another one called `y` which contains a random real value.
3. Swap the values of `x` and `y`.

Number-related types

Time to talk about **operators**. They are comparable to mathematical operations, yet in programming there is a larger variety of operators. To the basic 4 arithmetic operators (+, -, *, /), the **integer division (//)**, **modulo (%)**, **negation (-)** and **exponentiation (**)** are added.

```
x = 17
y = 5

# sum
print(x+y) # print() writes out whatever is inside the parentheses
# difference
print(x-y)
# product
print(x*y)
# division
print(x/y) # with decimals
# integer division
print(x//y) # without decimals
# modulo
print(x%y) # the remainder of x//y
# negation
print(-x) # changes the sign of x
# exponentiation
print(x**y) # x to the y-th power
```

22

12

85

3.4

3

2

-17

1419857

Number-related types

Assignment operators are used to associate values to variables. We've talked about the **basic assignment operator (=)**. It can also be combined with the binary operators.

Example:

```
x = 5 # x is 5  
x += 10 # equivalent to x = x+10  
# now x is 15
```

The expression $x = x + 10$ sounds like nonsense. Well, mathematically it does. In programming, it translates to *the new value of x is equal to the old value of $x + 10$* .

This can be done with any operator (**$-=$** , **$//=$** , **$**=$** etc.)

Boolean values

The next most important type of value is the **boolean** (logical type) which can either be **True** or **False**. Their equivalents in bit form are 0 for **False**, 1 for **True**.

This type of value is used with **logical operators**.

The first 3 logical operators we'll talk about are used only with booleans: **logical and**, **logical or** and the **logical not**.

x	y	x and y
True	True	True
True	False	False
False	True	False
False	False	False

Boolean values

x	y	x or y
True	True	True
True	False	True
False	True	True
False	False	False

x	not x
True	False
False	True

Boolean values

More logical operators! We've only looked at those which work with booleans, but there are many more which take in other types as well, but return a boolean: the **equality operator** (`==`) checks whether or not 2 values are equal; the **not equal operator** (`!=`) returns True when 2 values are different and False otherwise; the **less than operator** (`<`) checks if the first value is smaller than the second; the **greater than operator** (`>`) checks if it's larger and so on.

Example:

```
x = 6
y = 9
z = 6

# checks for equality
print(x == y)
print(x == z)

# are x and z different?
print(x != z)

# is x smaller than y?
print(x < y)

# is x larger than y?
print(x > y)

# is x smaller than or equal to z?
print(x <= z)

# is x larger than or equal to y?
print(x >= y)
```

```
False
True
False
True
False
True
False
```

Boolean values

Question time!

1. What does this expression equate to?


`(not (4 > 3)) or ((3.14 != 2.71 + 0.42) and False)`

2. Come up with an expression which evaluates to False.

Sequence types

What if you needed to store multiple values in the same place? Let's say you had to create a variable for each test grade in your class and then display them in ascending order. That would require tedious amounts of work! This is why **sequences** are useful.

```
li = [1, 2, 3.14, 4+2j, 5, 6.9, 7, 8]
```



list
declaration

item

The elements of a sequence are written inside **brackets** (either (...) or [...]) and separated by a **comma** (.). In Python, sequences can contain elements of different types simultaneously.

Sequence types

How do we access the elements of a sequence? Well, there are several methods. Let's take a look at our previous list.

```
li = [1, 2, 3.14, 4+2j, 5, 6.9, 7, 8]
# 0  1  2    3    4  5  6  7
# -8 -7 -6    -5   -4 -3 -2 -1
```

Each element of a sequence has an **index** corresponding to its position relative to the beginning of the list and the end of the list. Therefore, **the first element has its index at 0**, the second at 1 and so on. However, the elements can be counted backwards as well, with the **last element having an index of -1**, the second last -2 and so forth.

Sequence types

In order to access a certain element of a sequence we'll use the square bracket notation like so:

```
li = [1, 2, 3.14, 4+2j, 5, 6.9, 7, 8]
      # 0  1  2      3      4  5  6  7
      # -8 -7 -6      -5      -4 -3 -2 -1

print(li[3])
```

(4+2j)

Sequence types

The first type of sequence we'll cover is called the **range**. It represents the sequence of integers between 2 numbers (**which must also be ints**) in ascending order. Therefore, they are treated very much like constant values and cannot be changed. The range can take 2 arguments:

range(begin, end) contains the integers from begin to end-1

Its main purpose is to be iterated through as part of some **algorithms** we'll cover later on.

algorithm = a well-defined, finite set of instructions which can be universally used as a solution to a certain type of problem

Sequence types

Before moving on to the other 2 sequence types, we need to explain an important concept which is **mutability**.

A mutable sequence is a sequence whose elements **can be changed**. An **immutable** sequence is the contrary of a mutable one.

Even if a sequence is immutable, it can still be **extended** (but not **shrunk**) or **reassigned**.

The last 2 types of sequence we'll talk about are the **list** and the **tuple**.

The list is mutable and it uses **[...]** brackets, while the tuple is **immutable** and uses **(...)** parentheses to contain its elements.

Sequence types

List vs tuple comparison:

```
li = [1, 2, 3.14, 4+2j, 5, 6.9, 7, 8]

li[0] = 700 # changes element 1 with 700
li.remove(4+2j) # removes element 4+2j
li.insert(2, 1234) # inserts element 1234 after 3.14 and before 5
li.append(101) # inserts element 101 at the end of the list
li.reverse() # inverts the elements of li

print(li)
```

```
[101, 8, 7, 6.9, 5, 3.14, 1234, 2, 700]
```

```
tup = (1, 2, 3.14, 4+2j, 5, 6.9, 7, 8)

tup[0] = 700 # will raise a TypeError
tup.remove(4+2j) # can't do this
tup.insert(2, 1234) # can't do this either
tup.append(101) # nope
tup.reverse() # will raise yet another error

TypeError: 'tuple' object does not support item assignment
```

However, a common function they share is `len()` which returns the length of a

```
li = [1, 2, 3.14, 4+2j, 5, 6.9, 7, 8]
tup = (1, 2, 3, 4)

print(len(li)) # works
print(len(tup)) # also works
```

8

4

Sequence types

Sequences also support some operators (not as many as numericals do). For example the addition operator (+) from number types becomes the **concatenation operator (joins sequences together)**. It can be used between sequences of the same type only (list + list and tuple + tuple, not list + tuple).

```
li = [1, 2, 3]
tup1 = (4, 5, 6)
tup2 = (7, 8, 9)

li += [10, 100] # works
print(tup1 + tup2) # works as well
print(li)
```

(4, 5, 6, 7, 8, 9)

[1, 2, 3, 10, 100]

Sequence types

Finally, sequences use a new type of operators called the **membership operators**: the **in operator** and the **not in operator**. The first one returns True if a certain value can be found in a sequence, False otherwise. The latter does the contrary.

```
li = [1, 2, 3.14, 4+2j, 5, 6.9, 7, 8]
```

```
print(3.14 in li)  
print(4+5j in li)  
print(4+2j not in li)
```

```
True  
False  
False
```

Sequence types

Question time!

1. Suppose `li = [1, 2, 3, 4, 5, 6]`. What values do the following expressions take? `[2, 3, 4] in li`; `(2, 3, 4) in li`.
2. Given `tup = (1, [2, 3, 4], 5, 6)`, which of the following statements will raise an error? `tup[0] = 10`; `tup[1] = 0`; `tup[1].append(10)`; `tup[1][1] = 2`; `tup = (1, 2, 3, 4, 5, 6)`.
3. Perform the following operations on the tuple `s = (1, 2, 3, 4, 5, 6)`: remove 4, append 7, insert 9 as the second element and replace 3 with 10.

Text values

The next data type we'll cover takes in text values, called **strings**. These are treated like **immutable sequences of characters** delimited by **quotation marks** (“...” or ‘...’) instead of brackets. Observe the difference between these two strings which, theoretically, are composed of the same elements:

```
str1 = ("r", "a", "n", "d", "o", "m", " ", "s", "t", "r", "i", "n", "g") # a pain  
str2 = "random string" # much better
```

With single quotation marks we can create strings containing double quotation marks and vice versa

```
str1 = 'text inside "double quotes"'  
str2 = "text inside 'single quotes'"  
str3 = 'don't do this' # closes the string and creates a mess
```

Text values

You can't place every character directly into a string as some of them are considered **illegal** (take double quotes inside a double quoted string as an example). It would've been impossible to even place these at all had it not been for several special characters called **escape characters**. They always begin in a **backslash** (\) and the character right after it specifies which special character it represents.

Example:

```
print("HelloWorld") # no escape characters HelloWorld
print("Hello\nWorld") # \n = new line Hello
print("Hello\tWorld") # \t = tab World
print("\"HelloWorld\"") # \" = double quote Hello World
print('\'HelloWorld\'') # \' = single quote "HelloWorld"
print('\'HelloWorld\'') # \' = single quote 'HelloWorld'
```


Text values

Being immutable sequences, strings support square bracket notation, slicing, immutable sequence functions and operators. However, they can be modified using some string-specific functions which aren't found with other sequence types.

```
str1 = " Hello World "  
print(str1.lower()) # turns all characters lowercase  
print(str1.upper()) # turns all characters uppercase  
print(str1.strip()) # removes any leading and trailing whitespaces  
print(str1.replace("H", "j")) # replaces any instance of the 1st string with the 2nd  
print(str1.split(" ")) # returns a list of substrings separated by the character passed as argument
```

```
hello world  
HELLO WORLD  
Hello World  
jello World  
['', 'Hello', 'World', '']
```

Text values

You can easily concatenate 2 strings together using the + operator, but what about a string with another type? For example, adding together a string and an int will raise an error. For this reason, strings have a special function designed for multi-type concatenation.

```
name = "John"
grade = 8.5

str1 = "{}'s test grade is {}."
print(str1.format(name, grade)) # inserts the values of the arguments inside the formatting brackets in order

str2 = "{1}'s test grade is {0}."
print(str2.format(grade, name)) # inserts the values in the order specified inside the formatting brackets

str3 = "{n}'s test grade is {g}."
print(str3.format(n = name, g = grade)) # inserts the values specified by the keywords inside the brackets
```

```
John's test grade is 8.5.
John's test grade is 8.5.
John's test grade is 8.5.
```

Text values

Question time!

1. Find the length of the following string: `s = "this is a random /n sentence."`.
2. Suppose `x = 4`, `y = 3`, `z = 1`. Fill in the brackets with the right numbers so the string becomes "3.14": `"{}.{}{}".format(x, y, z)`
3. Using the functions explained previously, transform "Hello World" into "jejjo worjd".

User input

We've covered all the significant data types (yes, there are more, but they are of least concern right now). This begs the question:

“How do we get variables to store values as we (the users) wish?”.

Here is where `input()` comes into play. It is a function which returns whatever value the user entered by keyboard in the form of a string.

The `input` function is typically used as such:

```
x = input("Enter a number: ") # will print out whatever is passed as argument
print(x)
```

Enter a number: 37

Note: As a means of distinction, the console text is printed out in blue and the user prompt is written in black.

User input

Let's say we want another number y and add it to our initial one x . As we mentioned before, the input function returns only strings (we can check this using the `type(x)` function which returns the data type that the value of x is under), thus our addition operator will concatenate x and y .

```
x = input("Enter a number: ")
y = input("Enter another number: ")
print(type(x))
print(x+y)
```

Enter a number: 77
Enter another number: 33
<class 'str'>
7733

The solution for this is **typecasting**(converting a value from a certain type to another), done using several functions. In our case, the right function to use is `int()` (converts any value to int if possible)

User input

Example of typecasting:

```
x = input("Enter a number: ")
y = input("Enter another number: ")

print(type(x))
print(x+y)

x = int(x)
y = int(y)

print(type(x))
print(x+y)
```

Enter a number: 77
Enter another number: 33
<class 'str'>
7733
<class 'int'>
110

As casting can be done to any type we've covered in this presentation, we can talk about the following typecasting functions: `int()`, `float()`, `bool()`, `list()`, `tuple()`, `str()`. However, there are some restrictions to casting (for example, you cannot convert a numerical to a sequence or vice-versa, except for strings)

User input

Question time!

1. Read a float from the keyboard.
2. Remove the decimals in front of said number using the typecasting functions presented earlier.

Control structures

So far, the code snippets we have taken a look at had a **linear** structure (the statements are executed once and in the order they appear within the code). However, we can control the flow of our program using **control structures**.

Beside linear, there are 2 more types of structures: **decisional** (executes a block of code depending on a given condition) and **repetitive** (places a block of code inside a loop).

First of all, we'll cover decisional structures, which come into several types themselves.

Control structures

The simplest decisional structure is the **if-else** statement. As we've seen at the beginning of the presentation, the instruction blocks inside the if-else structure begin with a colon and respect the indentation rule. A very basic code snippet which makes use of this structure is **checking whether a number is even or odd**:

```
x = int(input("x = "))    x = 36
                          x is even.
if x%2 == 0:
    print("x is even.") =====
else:
    x = 37
    print("x is odd.")    x is odd.
```

Let's make sense of this code line by line. The first line generates an input whose value will be converted from string to int (so we could be able to use arithmetic operators on it) and finally storing it into x.

Note: The “===” line separates independent program executions.

Control structures

```
x = int(input("x = "))    x = 36
                           x is even.
if x%2 == 0:
    print("x is even.")   =====
else:
    print("x is odd.")    x = 37
                           x is odd.
```

Now onto the control structure. Before executing any instruction blocks, the condition will be evaluated. In the first example, x takes the value of 36. $36\%2$ will be equal to 0 as the division between the two leaves no remainder, thus the whole condition evaluates to **True** and the block between if and else will be executed (the console will print out “x is even.”) and **the other block will be skipped**. As for the second example, $x = 37$. $37\%2$ will equal 1, therefore the condition evaluates to **False**, causing the if block to be skipped and the else block to be executed.

Note: else doesn't have any conditions as it covers **any other case** than the if

case.

Control structures

What if you needed more conditions? That's no problem. The if-else statement has a more complex version of itself: the **if-elif-else** statement. The elif branch takes in an intermediate condition in case the if condition evaluates to False. Let's take a look at 2 examples:

```
x = int(input("x = "))
if x < 10:
    print("x has 1 digit.")
elif x >= 10 and x < 100:
    print("x has 2 digits.")
else:
    print("x has 3 or more digits.")
```

```
x = 3
x has 1 digit.

=====
x = 10
x has 2 digits.

=====
x = 314
x has 3 or more digits.
```

```
x = int(input("x = "))
r = x%4

if r == 0:
    print("x is divisible by 4.")
elif r == 1:
    print("x-1 is divisible by 4.")
elif r == 2:
    print("x-2 is divisible by 4.")
else:
    print("x-3 is divisible by 4.")
```

```
x = 16
x is divisible by 4.

=====
x = 17
x-1 is divisible by 4.

=====
x = 23
x-3 is divisible by 4.
```

Note: You can have as many elif branches as you like. You can join more conditions together on a single branch using the boolean operators and, or, not.

Control structures

What happens if 2 or more conditions in the same decisional structure evaluate to `True`?

```
x = int(input("x = "))  
  
if x%4 == 2:  
    print("x-2 is divisible by 4.")  
elif x%3 == 1:  
    print("x-1 is divisible by 3.")  
  
x = 10  
x-2 is divisible by 4.
```

```
x = int(input("x = "))  
  
if x%3 == 1:  
    print("x-1 is divisible by 3.")  
elif x%4 == 2:  
    print("x-2 is divisible by 4.")  
  
x = 10  
x-1 is divisible by 3.
```

As you can see, no matter how many conditions in the whole structure are fulfilled, only the first block gets executed and the others are skipped. This is why it's recommended to use different structures for checking unrelated conditions (like the divisors are different, as in the previous example).

```
x = int(input("x = "))  
  
if x%4 == 2:  
    print("x-2 is divisible by 4.")  
if x%3 == 1:  
    print("x-1 is divisible by 3.")  
  
x = 10  
x-2 is divisible by 4.  
x-1 is divisible by 3.
```

Note: Decisional blocks can be incomplete. The presence of branch always marks the beginning of a new block.

Control structures

There is no problem with using decisional blocks within other decisional blocks as in this code snippet which determines **the maximum of 3 numbers**:

```
a = int(input("a = "))
b = int(input("b = "))
c = int(input("c = "))

if a > b:
    if a > c:
        print("max = a")
    else:
        print("max = c")
elif b > c:
    print("max = b")
else:
    print("max = c")
```

Explanation: After reading the values of a, b and c, we'll do several comparisons. We're taking the first case, when $a > b$. If a is also greater than c, then a is the greatest number. Otherwise, $c \geq a$ (therefore $c \geq a > b$), meaning that the maximum is c. If the elif branch (b is greater than c) gets executed, then we can also deduce that b is also greater than or equal to a, making b the maximum. If we get to the else block, we'll have that $b \geq a$ and $c \geq b$ (written compactly as $c \geq b \geq a$), case in which c is the largest.

Control structures

The last type of decisional structure is called the **match-case** statement. It is highly similar to the if-elif-else block, except it has its differences: it is often used as a shorthand for the aforementioned structure in case it has too many elif branches and, also, it can be used only for certain values at a time (that is, the first example presented on Slide 48 can't be rewritten in match-case format, while the second can). Here is Example 2 from Slide 48 using a match-case block.

```
x = int(input("x = "))
r = x%4

match r: # the value we want to check is stored in r
    case 0: # equivalent to if r == 0:
        print("x is divisible by 4.")
    case 1:
        print("x-1 is divisible by 4.")
    case 2:
        print("x-2 is divisible by 4.")
    case other: # equivalent to else:
        print("x-3 is divisible by 4.")
```

Control structures

What if you wanted to repeat some sequence of instructions multiple times? The answer is simple: you'd place that instruction block inside a **repetitive structure**. There are 2 types of repetitive structures, both being used for different algorithms. For example, you'd want to use a **while loop** (executes a code block as long as some condition is true) for calculating the digit sum of a number.

```
x = int(input("x = "))
s = 0

while x > 0:
    s += x%10
    x //= 10

print(s)
```

```
x = 7312
13
```

Control structures

```
x = int(input("x = "))    x = 7312
s = 0                    13

while x > 0:
    s += x%10
    x //= 10

print(s)
```

Explanation: In the first line, the program is reading the value of x. Afterwards we're creating a variable in which to store the digit sum of x (initially set to 0). Moving on to the loop, we see that the condition under which the structure will be executing is $x > 0$. Why is that? As soon as x reaches 0 we will have run out of digits to extract. We will add the remainder of $x//10$ to s (multiples of 10 always end in 0, therefore the remainder is exactly the last digit of x). The variable x will then be set to $x//10$ (equivalent to removing its last digit, as integer division ignores the remainder and division by 10 is the same as deleting a trailing zero). This will be done until $x == 0$, time by which we had added together all the digits of x (but backwards).

Control structures

When iterating through a sequence the ideal structure to use is the **for** loop (creates a temporary variable which takes every value of a sequence in order and one at a time)

```
li = [6, 3, 9, 13, 7]
s = 0

for x in li:
    s += x

print("The sum of the elements in li is {}".format(s))

The sum of the elements in li is 38
```

```
# MAXIMUM OF N NUMBERS

n = int(input("Enter the number of elements: "))
elem = int(input())
nmax = elem

for i in range(n-1):
    elem = int(input())
    if elem > nmax:
        nmax = elem

print("The maximum number in this sequence is {}".format(nmax))
```

Enter the number of elements: 6
2
8
9
3
3
4
The maximum number in this sequence is 9

Note: You can use ranges to execute a loop for a definite number of times

Control structures

Need to stop the flow of a loop before the condition expires? Using the **break statement** will resume the execution by skipping to the first instruction from outside the loop

```
0
1
2
3
4
5
6
7
8
9
i = 0
while True: # indefinite loop
    if i == 10:
        break
    print(i)
    i += 1
```

What if you want the loop to skip a step in certain cases? There is the **continue statement** which restarts the loop and, in the case of for loops, updates the iterating variable to the next value.

```
5  for i in [5, 10, 15, 20, 25, 30, 35, 40, 45, 50]:
15     if i%10 == 0:
25         continue
35     print(i)
45
```

Functions

Some programs can take up to hundreds – even thousands – of lines and for this reason become incredibly difficult to read. In order to avoid repetitively writing the same blocks of code in order to execute one task in multiple instances, you can use **functions** (instruction sequences which execute only when called). Compare these 2 codes:

```
a = 1357
b = 123456

s1 = 0
s2 = 0

while a > 0:
    s1 += a%10
    a //= 10

while b > 0:
    s2 += b%10
    b //= 10

print("The digit sum of a is {0} and the one of b is {1}".format(s1, s2))
```

(Without functions)

```
def digitsum(n):
    s = 0
    while n > 0:
        s += n%10
        n //= 10
    return s

a = 1357
b = 123456

print("The digit sum of a is {0} and the one of b is {1}".format(digitsum(a), digitsum(b)))
```

(With functions)

Functions

Let's build our `digitsum()` function step by step. First, let's declare an empty function:

```
def digitsum():  
    pass
```

As you can see, the function declaration in Python is called **def**, followed by the name of the function and the parentheses (required for the arguments, even if there are none). Don't forget to type the colon! The **pass** statement in the function body acts like a **null statement (doesn't do anything)**. Note that leaving a function body entirely empty will result in an error. The same applies to other structures!

`def digitsum():`
 `print("Calculating the digit sum...")` ions instead of `pass`. Let's update our function

```
digitsum() # function call  
Calculating the digit sum...
```

Functions

Next up we'll talk about arguments. In our case, we need to calculate the digit sum *of something*. That something becomes our **argument**. The updated function will look like this:

```
def digitsum(n):  
    print("Calculating the digit sum of {}".format(n))  
  
digitsum(1357) # function call  
  
Calculating the digit sum of 1357...
```

Furthermore, besides executing instructions, our function has to evaluate to some value. For this, we'll use the **return** statement and we can finally plug in the

digit sum algorithm:

```
def digitsum(n):  
    s = 0  
    while n > 0:  
        s += n%10  
        n //= 10  
    return s
```

Functions

Beside carrying a set of instructions which execute upon calling, functions can also behave like objects. For example, if you wanted a function which would do *anything to a* number, you'd need a function object to tell it what exactly to do.

```
def performOperation(func, n):  
    return func(n)
```

```
def digitsum(n):  
    s = 0  
    while n > 0:  
        s += n%10  
        n //= 10  
    return s
```

```
def prime(n):  
    if n < 2:  
        return False  
    else:  
        i = 2  
        while i**2 <= n:  
            if n%i == 0:  
                return False  
            i += 1  
    return True
```

```
f = digitsum  
g = prime  
  
print(performOperation(f, 209))  
print(performOperation(g, 209))
```

In this case, we have 2 operations available to perform on any number: calculate the digit sum and check for primality

False if it finds a divisor of n less than or equal to

t – most efficient as opposed to checking all or equal to n, for i should be a divisor of n, so otherwise) and the function

tion(func, n), which applies one of these 2

ic on a number n. We assign f to be an function object

n (same reasoning to be applied in the case of g). We then

11

False operations f (calculate the digit sum) and g (check primality) on the number 209

Functions

Speaking of prime numbers, how would we generate the sequence of prime numbers using a function? Python comes in with a useful feature, the **yield** statement (equivalent to return, but also retains the state of the function exactly before finishing, thus the next call returns the value specified inside the following **yield statement**). With this in mind, let's generate the first 10 prime numbers using

```
def prime(n):
    if n < 2:
        return False
    else:
        i = 2
        while i**2 <= n:
            if n%i == 0:
                return False
            i += 1
        return True

def primeSeq():
    i = 2
    while True:
        if prime(i):
            yield i
        i += 1

f = primeSeq()

for i in range(10):
    print(next(f))
```

In this snippet, our generating function continues the sequence indefinitely. The variable f will

reference to the beginning of primeSeq() (the first printing the

ed by
ceed with

xt() function (returns the
is pointing to, then moves f one
())

Functions

What if we have an indefinite number of arguments so we can't list them all in the function definition? In Python, we can transform multiple arguments into a single one using the **packing/unpacking operator(*)**. Let's generalize the `sum(a, b)` function from Slide 3 by

using this feature:

```
def multipleSum(*args):  
    s = 0  
    for x in args:  
        s += x  
    return s  
  
print(multipleSum(3, 5))  
print(multipleSum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
```

8
55

The asterisk here takes all the arguments passed in the function and **packs them into a single tuple** called `args`. Now suppose you have a sequence (called `s`) of numbers whose sum you want to calculate. First you'll have to **unpack (deconstruct the list down to independent arguments)** using the same operator to avoid packing the whole list into the tuple (as in `([x1, x2, x3, ..., xn])`). Thus, the correct syntax is `multipleSum(*s)` as opposed to `multipleSum(s)`.

Functions

In order to make code readability easier, there is another packing/unpacking operator (******) which works with **keyword arguments** (dictionary-like structures which map values to keys) the same way ***** does with lists.

Example:

```
def drawSquare(size, **kwargs):
    for i in range(size):
        print(kwargs["border"], end = '')
    print('\n')
    for i in range(size-2):
        print(kwargs["border"], end = '')
        for i in range(size-2):
            print(kwargs["fill"], end = '')
        print(kwargs["border"], end = '')
        print('\n')
    for i in range(size):
        print(kwargs["border"], end = '')

drawSquare(10, border = '/', fill = '#')
```

Note: The ****** operator packs arguments into dictionaries, hence the notations. The argument order in a function call should look like this: **some_function(plain_args, *args, **kwargs)**. The **end** argument in **print()** is also a keyword argument which defaults to **'\n'** if not specified (this is why **print()** automatically creates a new line after it's done). Here the end character is empty (the next **print()** statement continues exactly where the previous one left off).

Turtle graphics

With all the knowledge accumulated along this course, our programs are still limited to console text. It's time to take things up a notch and begin using graphics. There are more **libraries (additions to the base language)** which implement graphics in Python, but one of the simplest ones is called **Turtle (the turtle is a cursor which leaves a trace along its trajectory)**. First we have to learn how to use `import turtle`

This instruction lets the program know that you're using the contents of the Turtle library. Without it Python wouldn't know where to find the objects and methods related to Turtle.

Turtle graphics

Now that we've imported the library, let's create our first Turtle object.

```
import turtle
```

```
t = turtle.Turtle() # creates an object belonging to the Turtle class
```

Upon creating this object, running the program won't get only the console to pop up, there will also be a screen on which the turtle will be “drawing”.

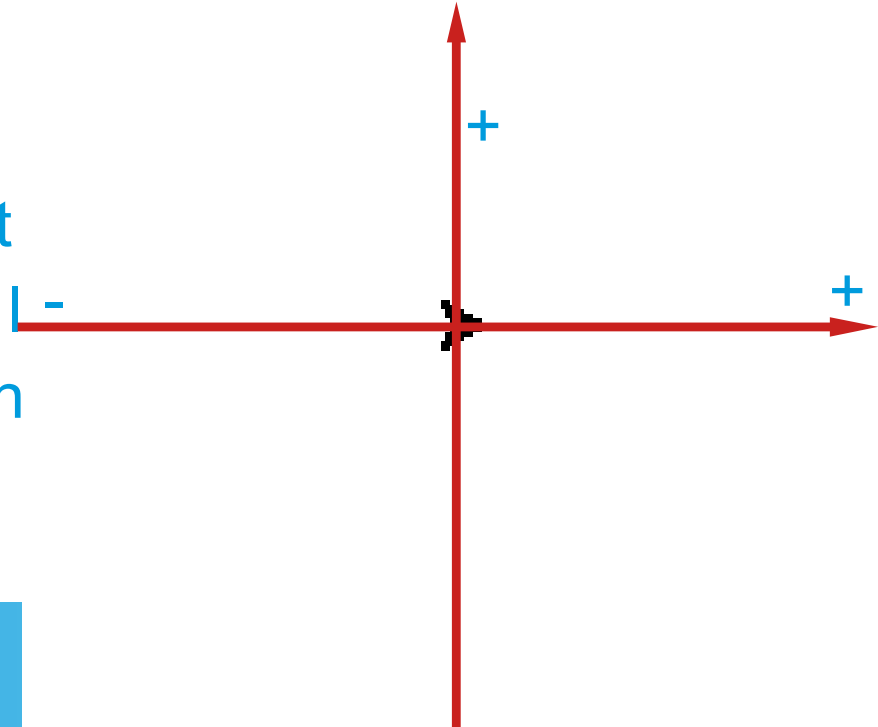
The turtle object is represented by the right-facing arrow.



Turtle graphics

We still haven't got the turtle to do anything, so the last picture showcases the **default position (Home)** of the turtle. This position has the coordinates $(0, 0)$ in the window

Moving the turtle x pixels away horizontally from this point and y pixels away vertically would get it at the point (x, y) . Notice that going to the left down would mean a negative value of pixels.



Turtle graphics

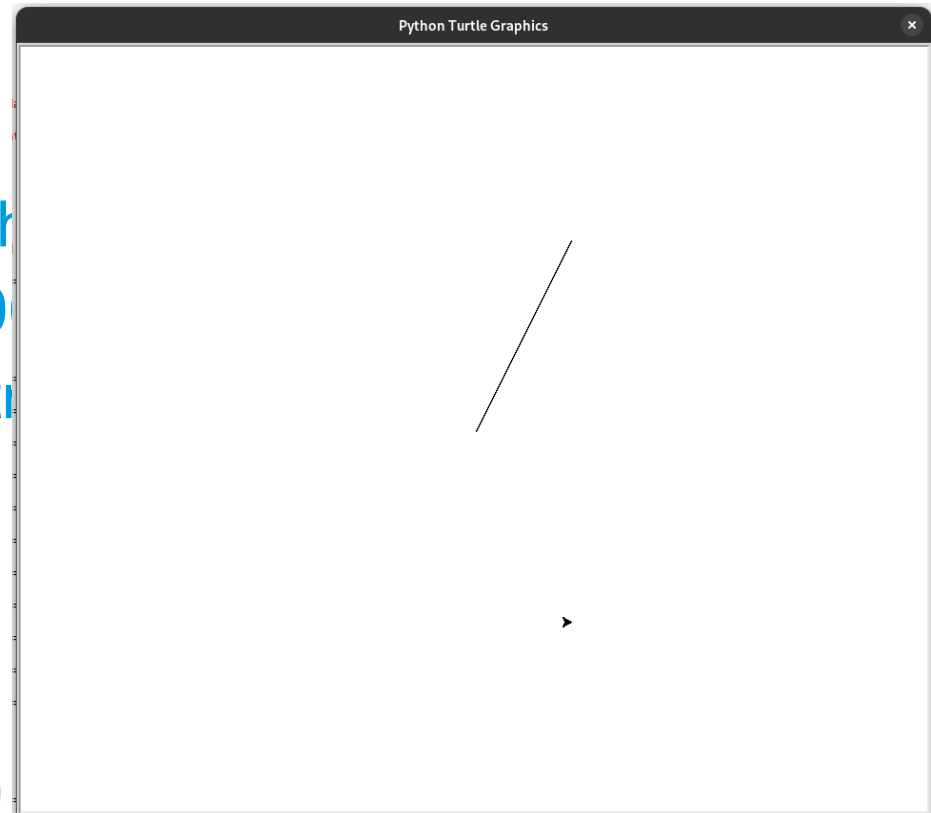
The turtle has 2 states: **pen up** (not leaving traces when moving) and **pen down** (draws its trajectory). By default the turtle's pen is down.

As we can see, the turtle first went to (100, 200) while drawing, then moved again to (100, -200) leaving a trace.

```
import turtle

t = turtle.Turtle()

# pen down
t.goto(100, 200) # goes to (100, 200)
t.penup() # lifts the pen
t.goto(100, -200) # goes to (100, -200)
```



Turtle graphics

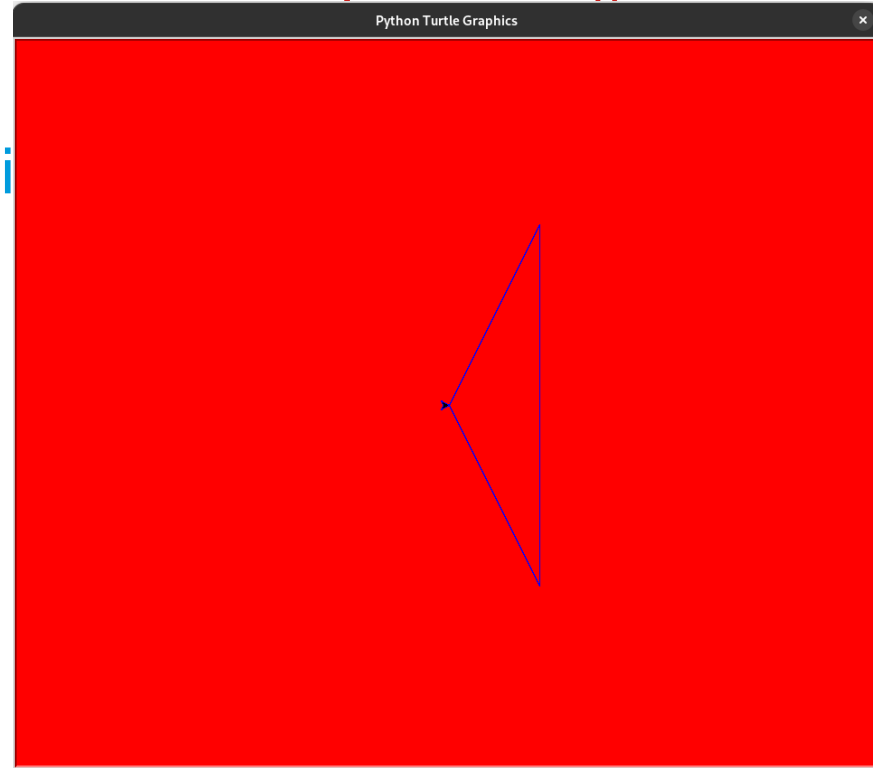
If we want to get the turtle back to (0, 0) we can use either `t.goto(0, 0)` or `t.home()`. We can change the background color to any color using `turtle.bgcolor()` and the turtle color with `t.pencolor()`

We can clearly see how the colors have changed and the turtle is now in home position.

```
import turtle

t = turtle.Turtle()

turtle.bgcolor("red")
t.pencolor("blue")
t.goto(100, 200)
t.goto(100, -200)
t.home()
```



Turtle graphics

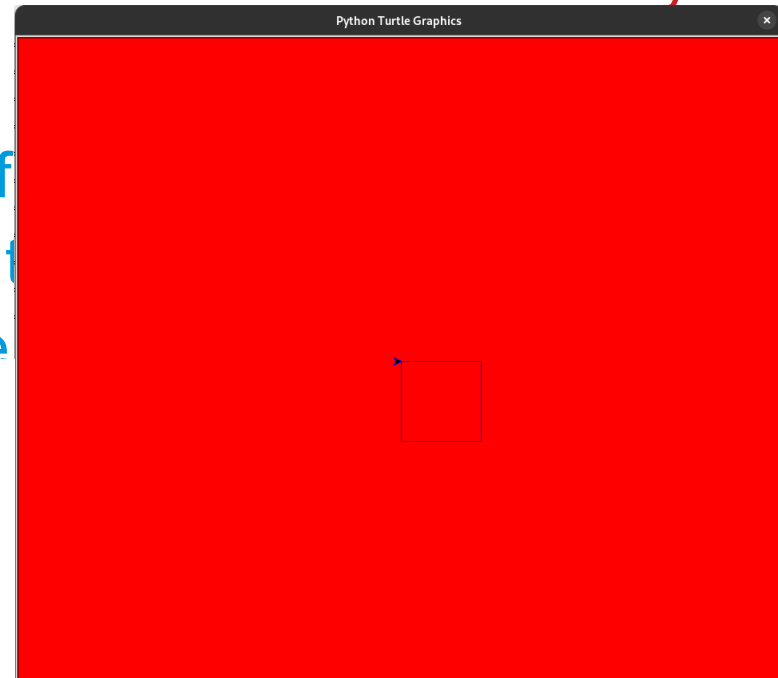
We've seen how the turtle is facing right by default. We can change this using `t.lt(x)` or `t.rt(x)` which change the orientation of the turtle to the left by x degrees and to the right, respectively. With the help of `t.fd(x)` and `t.bk(x)` (move the turtle forwards or backwards by x pixels) we can draw a square.

We've arbitrarily chosen a side length of 100 pixels. The turtle then turns to the right by 90 degrees (all the sides of a square are 90°)

has

```
import turtle
t = turtle.Turtle()

turtle.bgcolor("red")
t.pencolor("blue")
for i in range(4):
    t.fd(100)
    t.right(90)
```



Turtle graphics

How can we draw any regular polygon? Let's do a little bit of math: in order to draw the polygon, the sum of the angles of each turn must add up to 360° . Therefore, for a regular n -gon, the turtle must steer right by $360/n^\circ$ each time. Our program would look like

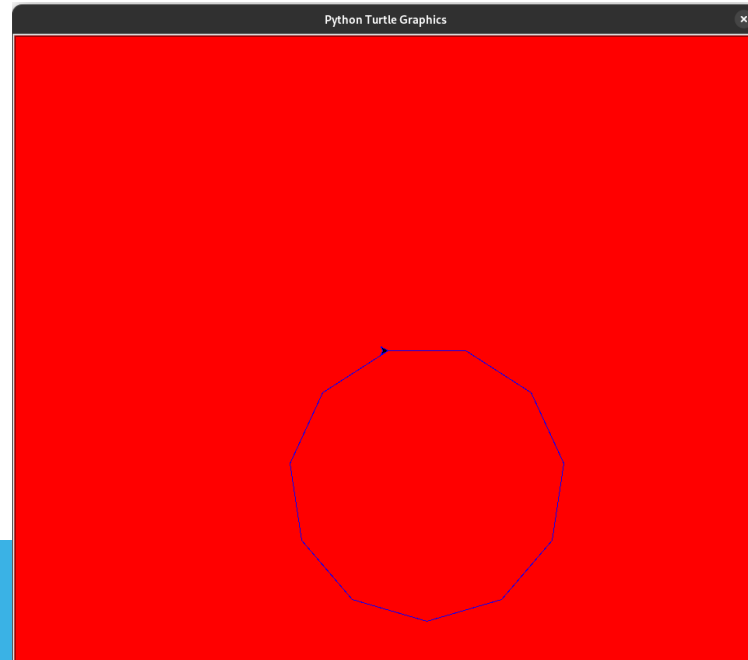
```
import turtle

n = int(input("Enter the number of sides: "))

t = turtle.Turtle()

turtle.bgcolor("red")
t.pencolor("blue")
for i in range(n):
    t.fd(100)
    t.right(360/n)
```

Enter the number of sides: 11



Turtle graphics

We can also fill a closed shape using whatever color we want using 3 commands: `t.fillcolor()`, `t.begin_fill()` and `t.end_fill()`

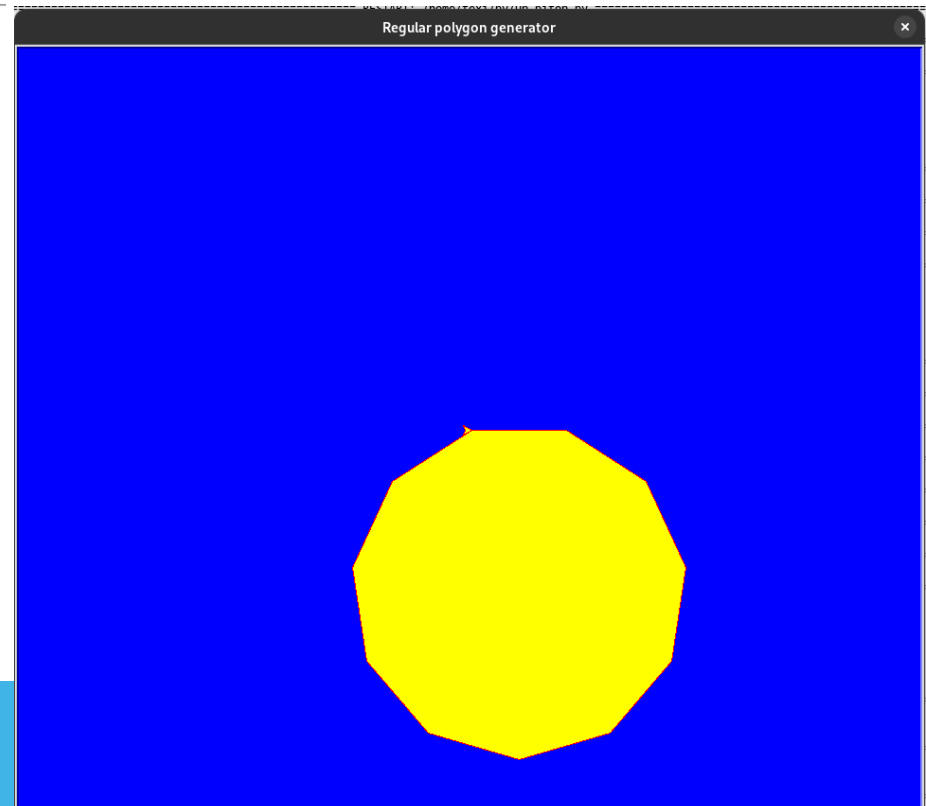
```
import turtle

n = int(input("Enter the number of sides: "))

t = turtle.Turtle()

turtle.title("Regular polygon generator") # changes the title of the Turtle window
turtle.bgcolor("blue")
t.pencolor("red")
t.fillcolor("yellow") # what color to fill the shape with
t.begin_fill() # the shape to be filled begins to be drawn from here
for i in range(n):
    t.fd(100)
    t.right(360/n)
t.end_fill() # the shape to be filled ends here, fills shape
```

Enter the number of sides: 11



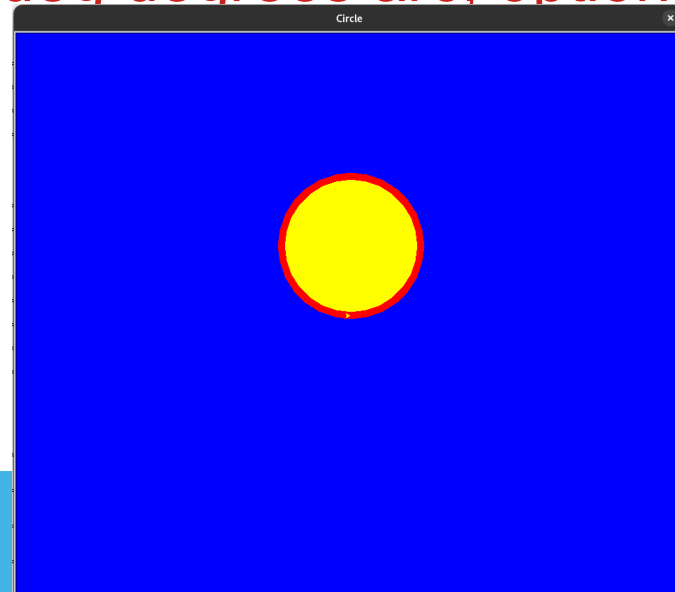
Turtle graphics

Feel like the contour of the shape is too thin? Use `t.pensize(x)` to set the line thickness to `x` pixels. Let's draw a circle this time. Using the technique for polygons, it would take tedious amounts of time. Fortunately, Turtle has an in-built function for drawing a circle: `t.circle(x, deg)` (radius of `x` pixels and `deg` degrees arc, optional: defaults to 360 if unspecified)

```
import turtle

t = turtle.Turtle()

turtle.title("Circle")
turtle.bgcolor("blue")
t.pencolor("red")
t.fillcolor("yellow")
t.pensize(10)
t.begin_fill()
t.circle(100)
t.end_fill()
```



Turtle graphics

We can draw more complex curves using the circle function. The following code snippet draws the Fibonacci spiral:

```
import turtle

def fib(n):
    if n in (1, 2):
        return 1
    else:
        return fib(n-1) + fib(n-2)

t = turtle.Turtle()

turtle.title("Fibonacci spiral")
turtle.bgcolor("blue")
t.pen(pencolor = "red", fillcolor = "yellow", pensize = 10) # compacts the attributes of the turtle into one function, optimal if you have multiple turtles
```

```
for i in range(1, 16):
    t.circle(fib(i), 90)
```



Turtle graphics

Another important function of a turtle is cloning. Using `t.clone()` will assign a Turtle object to a variable which is in the same state as `t`. With this function, we can draw a tree using recursion. Notice the speed attribute was changed so the tree would render faster.

```
import turtle

def tree(t, ht, wd, deg, n): # t = turtle object, ht = length of each branch, wd = branch width, deg = deviation angle of the clones, n = number of iterations
    if n > 0:
        t.pensize(wd)
        t.fd(ht)

        x = t.clone()
        y = t.clone()

        x.lt(deg)
        y.rt(deg)

        tree(x, ht, wd/1.5, deg*1.5, n-1)
        tree(y, ht, wd/1.5, deg*1.5, n-1)

t = turtle.Turtle()

turtle.title("Tree of Life")
turtle.bgcolor("blue")
t.pen(pencolor = "brown", fillcolor = "green", speed = 10)

t.lt(90)

tree(t, 50, 25, 9, 7)
```

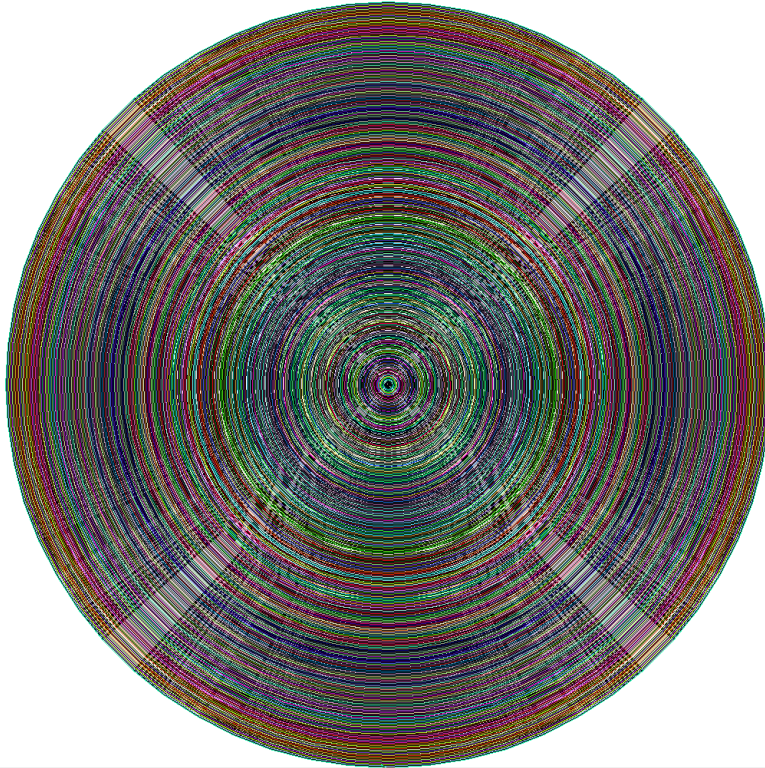


Turtle graphics

We've seen that `t.pencolor()` accepts color names as an argument, but how can we render more diverse colors? Here's where **hex codes** come into play. They represent a color using the **#RRGGBB** format (**R = amount of red**, **G = amount of green**, **B = amount of blue**), where 00 means the least color and FF means the most. This means that **#000000** represents black (no color at all) and **#FFFFFF** is white (all colors added). From black to white we can render 16777216 different colors. Example:

Turtle graphics

Final drawing



```
import turtle

t = turtle.Turtle()
t.speed(1000)

turtle.title("Final drawing")

for i in range(500): # draw 500 telescoping circles
    t.pencolor("#{:06X}".format(pow(i, i, 16777216))) # transform decimal number (must be less than 16777216) into 6-digit hexadecimal; for color "randomness" (i**i)%16777216 was chosen
    t.circle(i) # draw the circle

    # going 1 pixel outwards for the next circle
    t.rt(90)
    t.fd(1)
    t.lt(90)
```