Astroinformatics I (Semester 1 2025)

Programming Introduction

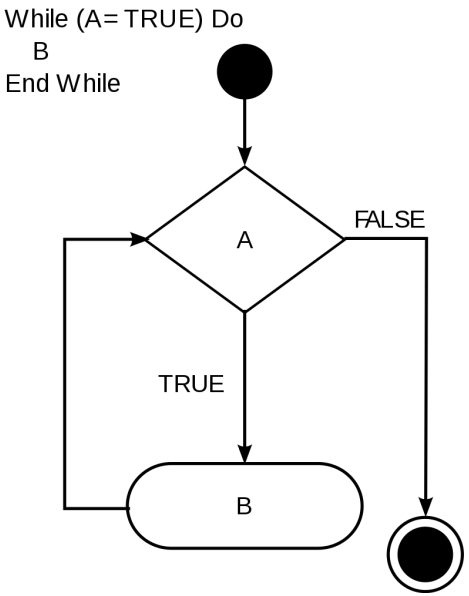
Nina Hernitschek
Centro de Astronomía CITEVA
Universidad de Antofagasta

May 12, 2025

Motivation

We have seen how to write simple shell commands.

To put those commands together into shell scripts, we will use a variety of new concepts such as **variables** and **control flow statements**, as well as more complex **algorithms**.
Many of those concepts also work well for other **programming languages**.



Writing a Program

Definition

A program is a self-contained list of commands that are stored in a file that can be read by an **interpreter** or a **compiler**.

Writing a Program

Definition

A program is a self-contained list of commands that are stored in a file that can be read by an **interpreter** or a **compiler**.

In the case of a shell script, it is a text file, with each line being the exact syntax you would have typed into the terminal.

Writing a Program

Computer programming is the process of designing and writing computer programs.

As a skill set, it includes a wide variety of different tasks and techniques.

This course is meant to provide basic, practical skills to help you understand and write computer code that helps you solving research questions in astronomy.

Programming Languages

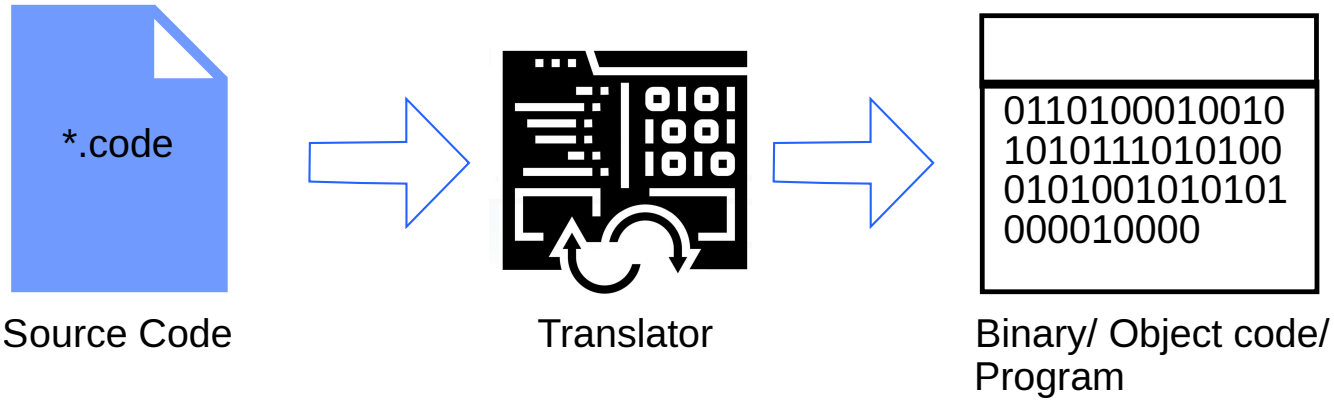
Internally, computers use binary code. Programming happens in languages that are closer to our own natural language: programming languages.

The most obvious difference between natural languages and programming languages is that programming languages are more structured and must be thoroughly learned.

Programming languages can be distinguished in **high level** and **low level languages**. High level programming languages generall offer a igher level of abstraction and are closer to natural languages. Low level programming languages are closer to the machine language.

Programming Languages

The **source code** we write must be converted into machine language.

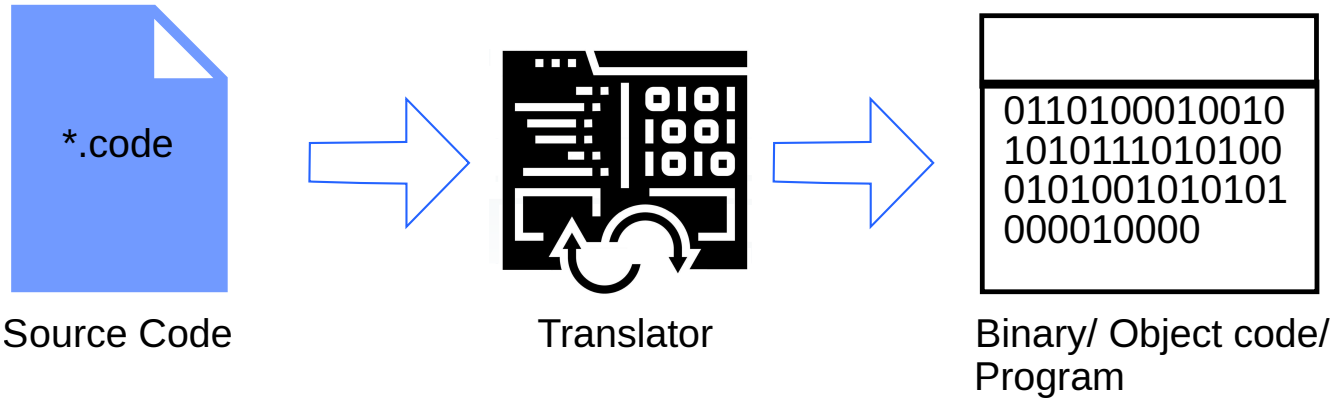


This can happen by:

- Interpreters
- Compilers
- a hybrid of Interpreters and Compilers
- Assemblers

Programming Languages

The **source code** we write must be converted into machine language.



This can happen by:

- Interpreters
- Compilers
- a hybrid of Interpreters and Compilers
- Assemblers

With using shell scripts, you have already worked with an interpeted script language.

Programming Languages

Interpreters

Some languages are interpreted. The interpreter processes the source code line by line to run the program. This means that interpreted source code starts running until it encounters an error. Then the interpreter stops to report such errors.
Examples of interpreted languages: bash, Python.

Compilers

Differently than interpreters that translate line by line and execute the line before going on to the next line, compilers translate all lines of a program to a file (binary) and execute the whole file. If there were errors in the source code, they are detected during the compilation time and flagged. This interrupts the compilation process, and no binary is generated.
Examples of compiled languages: C/C++, Fortran.

Hybrid Translators

A hybrid translator is a combination of the Interpreter and Compiler. A popular hybrid programming language is Java. Java first compiles your source code to an intermediate format known as the Bytecode. The Bytecode is then interpreted and executed by a runtime engine also known as a Virtual machine. This enables the hybrid translators to run the bytecode on various operating systems.

Assemblers

Assemblers translate low-level Assembly language to binary. Despite widely used until the 1980s, they aren't common anymore.

Programming Languages

A programming language is a set of instructions and syntax used to create software programs. Each language has its own strengths and weaknesses and is suited for different types of projects.

Some of the key features of programming languages include:

Syntax: The specific rules and structure used to write code.

Data Types: The type of values that can be stored in a program, such as numbers, strings, and booleans.

Variables: Named memory locations that can store values.

Operators: Symbols used to perform operations on values, such as addition, subtraction, and comparison.

Control Structures: Statements used to control the flow of a program, such as if-else statements and loops.

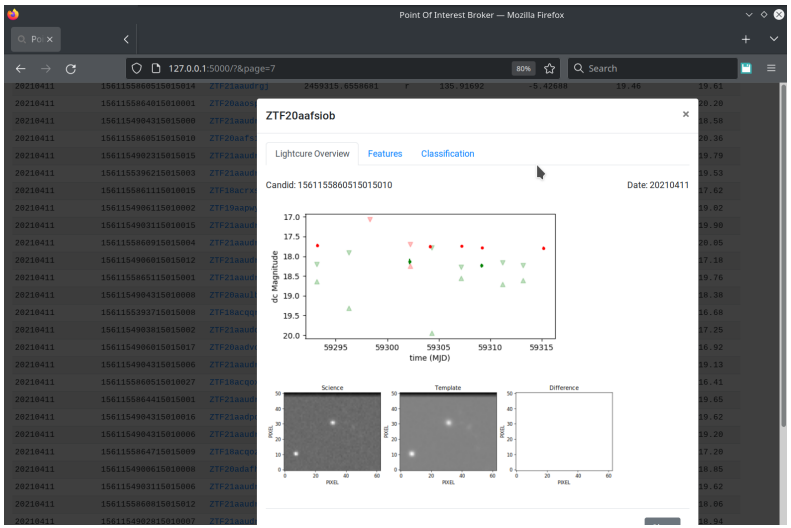
Libraries and Frameworks: Collections of pre-written code that can be used to perform common tasks and speed up development.

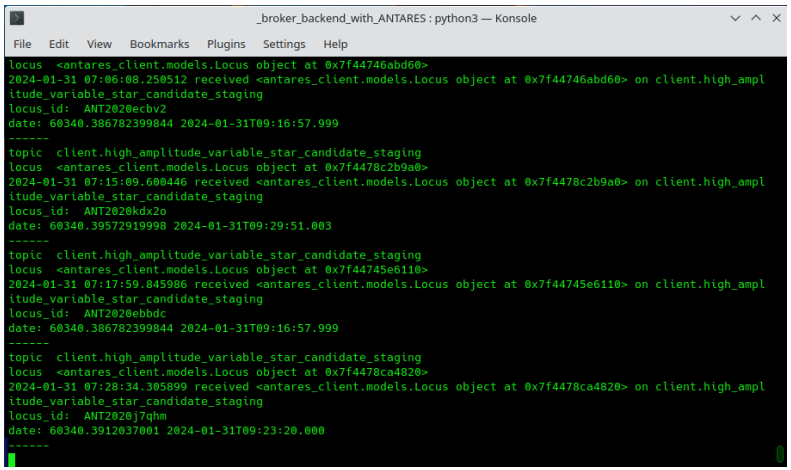
Paradigms: The programming style or philosophy used in the language, such as procedural, object-oriented, or functional.

Programming Languages

The choice of programming language depends on the specific requirements of a project, including the platform being used, existing software, libraries to use, the intended audience of the program (GUI required? should it run on the web?), and the desired outcome.

Programming languages continue to evolve and change over time, with new languages being developed and older ones being updated to meet changing needs.





left: a front-end coded in Python with the web framework Flask
right: the console output of a back-end coded in Python and C

Variables

A variable is where a program stores information:

Definition

A variable is a user-defined, symbolic name which points to a spot in a computer's memory where a value has been stored. The variable's name can then be used to retrieve the value, and the value can be changed at will.

Variables

A variable is where a program stores information:

Definition

A variable is a user-defined, symbolic name which points to a spot in a computer's memory where a value has been stored. The variable's name can then be used to retrieve the value, and the value can be changed at will.

Declaring variables is easy. In most programming languages it is done like the following:

```
x = 5.0
y = 'cat'
```

Variables

A variable is where a program stores information:

Definition

A variable is a user-defined, symbolic name which points to a spot in a computer's memory where a value has been stored. The variable's name can then be used to retrieve the value, and the value can be changed at will.

Declaring variables is easy. In most programming languages it is done like the following:

```
x = 5.0
y = 'cat'
```

Some programming languages distinguish between **variables and constants**:

Variables are mutable - they can be changed during program execution time. Constants are defined once and then cannot be changed.

Data Types

Programming languages usually have **data types** for all their variables.

Data Types

Programming languages usually have **data types** for all their variables.

Definition

Data types are the fundamental building blocks of a code, a property that every object/element/variable in a written code will have, and which will determine the rules by which a programming language operates on them.

Data Types

Programming languages usually have **data types** for all their variables.

Definition

Data types are the fundamental building blocks of a code, a property that every object/element/variable in a written code will have, and which will determine the rules by which a programming language operates on them.

Some of these different data types seem obvious: clearly a word like `"star"` is a fundamentally different data type than an list of numbers `[1,2,3,4,5]`.

Data Types

Programming languages usually have **data types** for all their variables.

Definition

Data types are the fundamental building blocks of a code, a property that every object/element/variable in a written code will have, and which will determine the rules by which a programming language operates on them.

Some of these different data types seem obvious: clearly a word like `"star"` is a fundamentally different data type than an list of numbers `[1,2,3,4,5]`.

Other divisions seem more arbitrary at first glance: For example many programming languages make the distinction between integers (the counting numbers), and floats (numbers with decimals). They do so because of the way computer processors store information in bits, and different data types take up different amounts of computer memory.

Data Types

Among the data types supported, we generally distinguish:

Primitive data types are those that are integrated into the programming language and do not require a previous definition.
These data types are found in most programming languages and can vary from one language to another, but generally include the following:

Integers: The counting numbers. Examples: `-1`, `0`, `1`, `2`, `3`, `4`, `5`, ...

Floats: Decimal numbers. Examples: `1.`, `2.345`, `6.3`, `999.99999`, ...

Strings: An iterable data type most commonly used to hold words/phrases or path locations. Denoted by single or double quotes. Examples: `"cat"` , `"/home/username/directory"`, `"1530"`, ...

Boolean: A data type with only two possible values: `True`, or `False`. They are used in conditional statements.

Data Types

Composite Data Types are those that contain more than one value. These types can be defined by the programmer or integrated into the programming language.
Composite data types offered by many programming languages are Lists and Arrays: They store any combination of data types, denoted with brackets.

While most programming languages only allow for all elements within a list or an array to be from the same type, others (such as Python) allow them to be mixed.

Examples: `[1,2,3,8]` or `[1, 2, [3, 4, 5], 'star']` (notice that you can have lists within lists)

Data Types

Reference Data Types are used for indirect data manipulation stored in other variables. These data types do not directly store the value, but refer to the memory location where the value or function is located.

A reference to a function is a data type that allows storing the memory address of a function. This allows for advanced structures within a program.

Reference data types are common in programming languages such as C++.

Data Types

In programming, there are two types of languages: typed languages and untyped languages.

Data Types

In programming, there are two types of languages: typed languages and untyped languages.

In a **typed language**, also called strictly typed, it is required to define the specific data type of the variables and expressions that we are going to use.

Examples of typed languages are C++, C#, and Java.

Data Types

In programming, there are two types of languages: typed languages and untyped languages.

In a **typed language**, also called strictly typed, it is required to define the specific data type of the variables and expressions that we are going to use.

Examples of typed languages are C++, C#, and Java.

On the other hand, **untyped languages**, also called dynamically typed, do not require us to specify the type of the variables and expressions when declaring the variable.

Examples of untyped languages are Python and JavaScript.

In reality, internally untyped languages do have types. Your program needs it to know how to manipulate the values of the variables. But the programming language makes its use transparent to the user most of the time.

Data Types in bash

In bash, there is no explicit data type and there is no explicit declaration of variables.
A variable can contain anything (including nothing).

A simple assignment is sufficient to create it:

```
a=1
```

The content of a variable can be obtained by prefixing it with $:

```
echo $a
```

Operators

Operators are symbols or keywords that allow different actions, such as performing arithmetic operations, comparing values, or assigning values to variables.

Operators are used to build expressions.

Operators

Operators are symbols or keywords that allow different actions, such as performing arithmetic operations, comparing values, or assigning values to variables.

Operators are used to build expressions.
An **expression** is a combination of values, variables, constants, operators, and functions that are evaluated to obtain a result.
In other words, expressions are the way blocks of code are constructed to perform calculations or manipulate data.

Operators

Operators are symbols or keywords that allow different actions, such as performing arithmetic operations, comparing values, or assigning values to variables.

Operators are used to build expressions.
An **expression** is a combination of values, variables, constants, operators, and functions that are evaluated to obtain a result.
In other words, expressions are the way blocks of code are constructed to perform calculations or manipulate data.

Expressions can be arithmetic, relational, logical, or other types, depending on the purpose to be achieved. Let's see some examples of expressions,

Some examples to illustrate this:

```
x = 5 + 3  ## Arithmetic expression

y = (x > 7)  ## Relational expression

z = (x > 7) and (y == True)  ## Logical expression
```

Arithmetic Operators

Arithmetic operators are used to perform **basic mathematical operations**, such as addition, subtraction, multiplication, and division.

The most common arithmetic operators are:

Operator	example	result
Addition (+)	`1 + 2`	3
Subtraction (-)	`10 - 3`	7
Multiplication (*)	`4 * 5`	20
Division (/)	`15 / 3`	5
Modulus (%)	`7 % 2`	1
Increment (++)	`a = 5; a++`	6
Decrement (--)	`b = 8; b--`	7

Arithmetic Operators

Arithmetic operators are used to perform **basic mathematical operations**, such as addition, subtraction, multiplication, and division.

The most common arithmetic operators are:

Operator	example	result
Addition (+)	`1 + 2`	3
Subtraction (-)	`10 - 3`	7
Multiplication (*)	`4 * 5`	20
Division (/)	`15 / 3`	5
Modulus (%)	`7 % 2`	1
Increment (++)	`a = 5; a++`	6
Decrement (--)	`b = 8; b--`	7

Caution: In many languages you will get different results when performing operations with whole numbers or decimal numbers.

Arithmetic Operators

Arithmetic operators are used to perform **basic mathematical operations**, such as addition, subtraction, multiplication, and division.

The most common arithmetic operators are:

Operator	example	result
Addition (+)	`1 + 2`	3
Subtraction (−)	`10 - 3`	7
Multiplication (*)	`4 * 5`	20
Division (/)	`15 / 3`	5
Modulus (%)	`7 % 2`	1
Increment (++)	`a = 5; a++`	6
Decrement (−−)	`b = 8; b--`	7

Caution: In many languages you will get different results when performing operations with whole numbers or decimal numbers.

Exponentiation is not common as an operator. When it is, it is usually represented by ^ or **. However, in most languages, exponentiation is usually a function in some library. The same is for trigonometric functions.

Comparison Operators

Programming languages also provide comparison operators which are symbols or combinations of symbols that allow us to **compare two values** and determine if a certain condition is true or false. These operators are widely used in **control structures** (e.g.: conditionals, loops).

Comparison operators return a Boolean value.

The most common comparison operators are:

Operator	example	result
Equal (==)	`a==a`	`True`
Not equal (!=)	`1!=2`	`True`
Greater than (>)	`10 > 5`	`True`
Less than (<)	`7 < 8`	`True`
Greater than or equal to (>=)	`5 >= 5`	`True`
Less than or equal to (<=)	`4 <= 4`	`True`

Comparison Operators

Programming languages also provide comparison operators which are symbols or combinations of symbols that allow us to **compare two values** and determine if a certain condition is true or false. These operators are widely used in **control structures** (e.g.: conditionals, loops).

Comparison operators return a Boolean value.

The most common comparison operators are:

Operator	example	result
Equal (==)	`a==a`	`True`
Not equal (!=)	`1!=2`	`True`
Greater than (>)	`10 > 5`	`True`
Less than (<)	`7 < 8`	`True`
Greater than or equal to (>=)	`5 >= 5`	`True`
Less than or equal to (<=)	`4 <= 4`	`True`

Caution: In most programming languages, a single = sign is reserved for setting the values of variables.

Logic Operators

Logical operators are used to perform **boolean operations** on values and variables. These operators are used in conditional structures and loops to **combine multiple conditions** and obtain a final result.

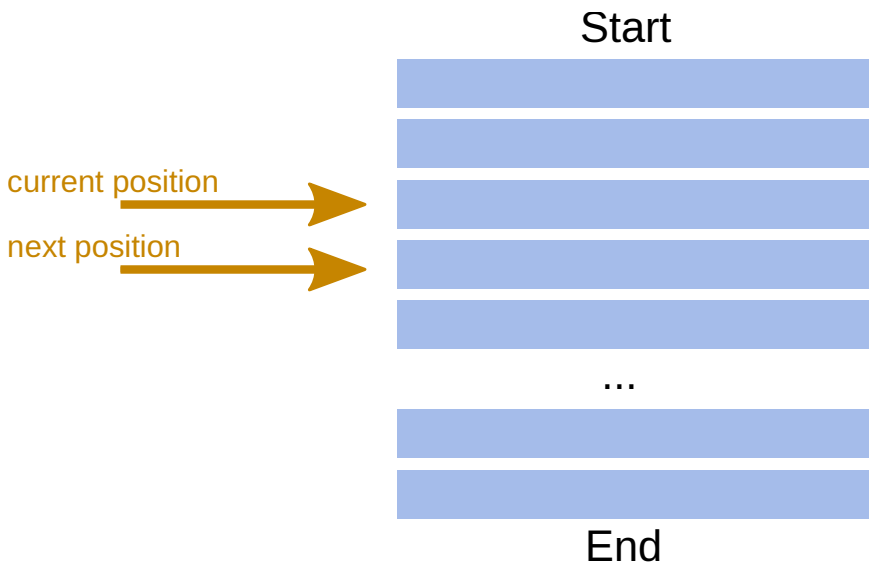
The most common logical operators are AND, OR, and NOT:
AND returns True if both operands are True, and False otherwise.
OR returns True if at least one of the operands is True.
NOT inverts a logical value.

We can show their results in a **truth table**:

A	B	A AND B	A OR B	NOT A
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

Control Flow Statements

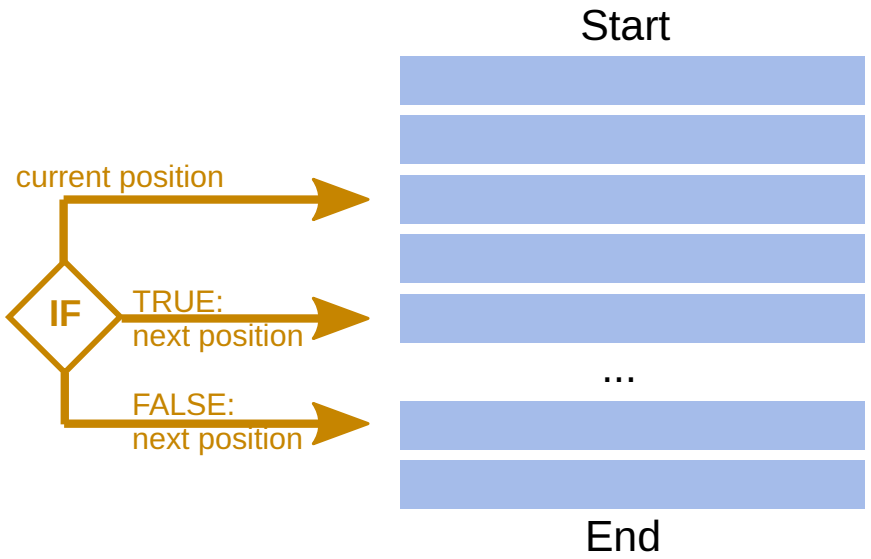
So far, we have shell scripts that are interpreted line by line in the order of their line number: sequential commands.



Control Flow Statements

So far, we have shell scripts that are interpreted line by line in the order of their line number: sequential commands.

The real power of programming, however, lies in our ability to write programs that don't just contain a list of sequential commands but which execution depends on various inputs. This is done by **control flow statements**.



Control Flow Statements

There are several types of flow control structures, including:

Jump structures: allow the programmer to transfer flow control to a different location in the program. The most common jump structure is the GO-TO. (Despite they have been standard, they should nowadays be avoided by all means.)

Control Flow Statements

There are several types of flow control structures, including:

Jump structures: allow the programmer to transfer flow control to a different location in the program. The most common jump structure is the GO-TO. (Despite they have been standard, they should nowadays be avoided by all means.)

Conditional statements: allow the program to make decisions based on a condition. The most common conditional structures are IF, IF-ELSE, IF-ELSEIF, and the ternary operator.

Control Flow Statements

There are several types of flow control structures, including:

Jump structures: allow the programmer to transfer flow control to a different location in the program. The most common jump structure is the GO-TO. (Despite they have been standard, they should nowadays be avoided by all means.)

Conditional statements: allow the program to make decisions based on a condition. The most common conditional structures are IF, IF-ELSE, IF-ELSEIF, and the ternary operator.

Loop structures: allow the program to repeat a sequence of instructions several times. The most common loop structures are WHILE, DO-WHILE, FOR, FOREACH.

Control Flow Statements

There are several types of flow control structures, including:

Jump structures: allow the programmer to transfer flow control to a different location in the program. The most common jump structure is the GO-TO. (Despite they have been standard, they should nowadays be avoided by all means.)

Conditional statements: allow the program to make decisions based on a condition. The most common conditional structures are IF, IF-ELSE, IF-ELSEIF, and the ternary operator.

Loop structures: allow the program to repeat a sequence of instructions several times. The most common loop structures are WHILE, DO-WHILE, FOR, FOREACH.

Error management structures: allow the program to respond to errors or exceptions. The most common one is TRY-CATCH.

Control Flow Statements

There are several types of flow control structures, including:

Jump structures: allow the programmer to transfer flow control to a different location in the program. The most common jump structure is the GO-TO. (Despite they have been standard, they should nowadays be avoided by all means.)

Conditional statements: allow the program to make decisions based on a condition. The most common conditional structures are IF, IF-ELSE, IF-ELSEIF, and the ternary operator.

Loop structures: allow the program to repeat a sequence of instructions several times. The most common loop structures are WHILE, DO-WHILE, FOR, FOREACH.

Error management structures: allow the program to respond to errors or exceptions. The most common one is TRY-CATCH.

Here, we will only look into conditional statements and loops.

Conditional Statements

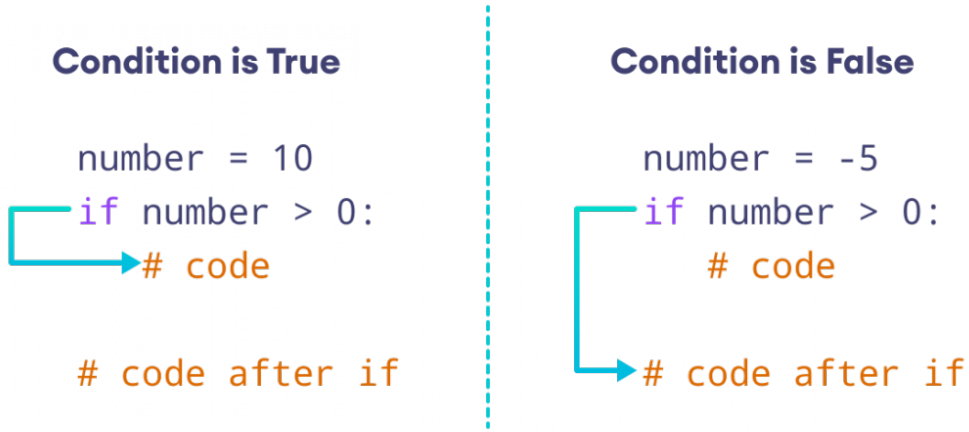
Conditional statements allow us to create branches in the program flow, executing certain instructions based on the evaluation of one or more conditions that are either True or False.

Definition

A conditional statement begins a defined, separated block of code which only executes (runs) if the conditional statement is evaluated by the interpreter to be true. Essentially, you are telling the computer to *only run this block of code IF some condition is true*. The condition itself is determined by the programmer.

Conditional Statements

The `if` statement evaluates `condition`:



Conditional Statements

An `if` statement can have an optional **else** clause.

The `if...else` statement evaluates the given condition:

Condition is True

```
number = 10
if number > 0:
    # code

else:
    # code

# code after if
```

Condition is False

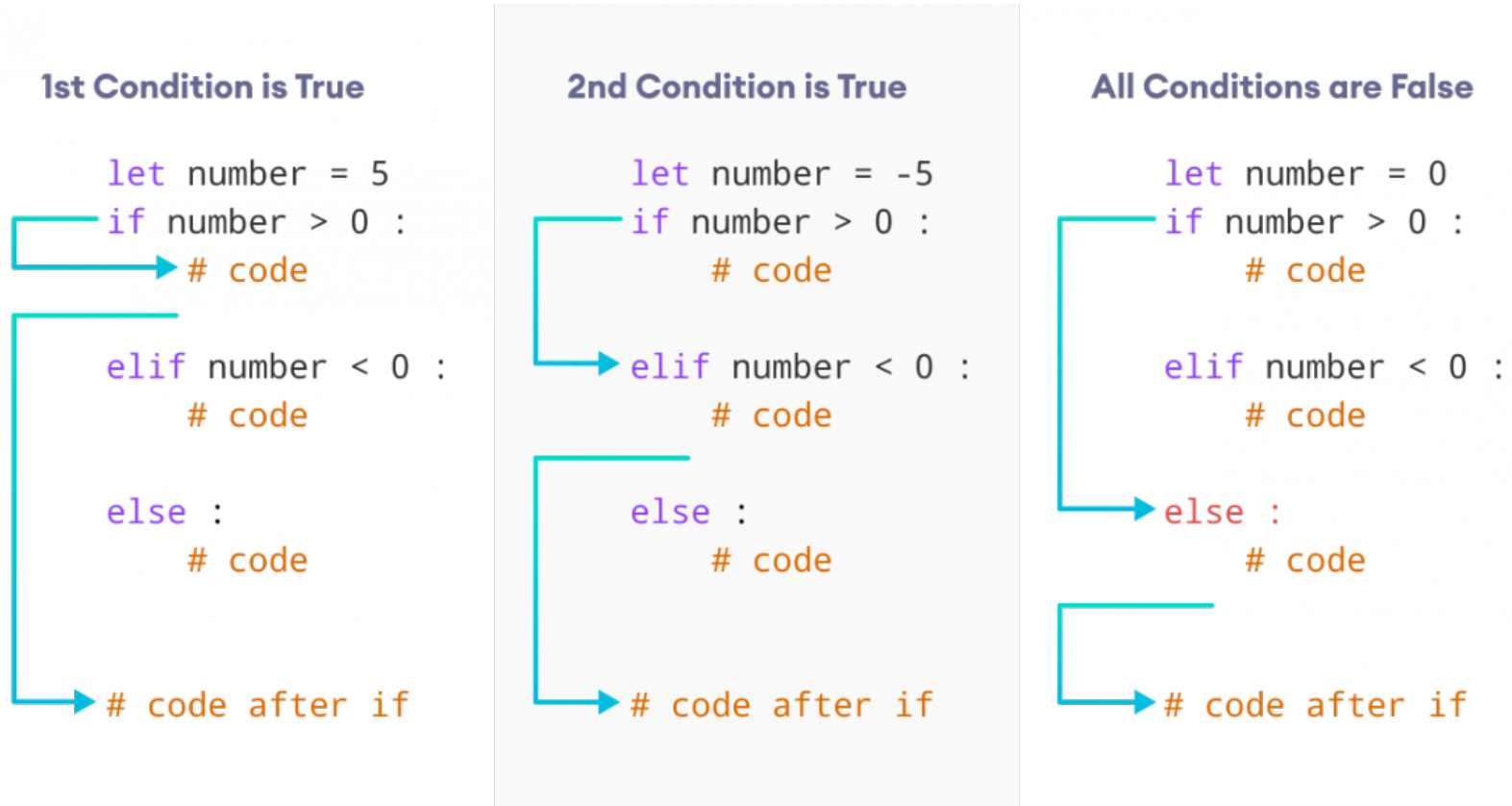
```
number = -5
if number > 0:
    # code

else:
    # code

# code after if
```

Conditional Statements

The `if...else` statement is used to execute a block of code among two alternatives.
However, if we need to make a choice between more than two alternatives, then we use the `if...elif...else` statement.



Conditional Statements

We can also use an `if` statement inside of an `if` statement. This is known as a **nested `if` statement**.

The syntax is as follows:

```
# outer if statement
if condition1:
    # statement(s)

    # inner if statement
    if condition2:
        # statement(s)
```

Conditional Statements

We can also use an `if` statement inside of an `if` statement. This is known as a **nested `if` statement**.

The syntax is as follows:

```
# outer if statement
if condition1:
    # statement(s)

    # inner if statement
    if condition2:
        # statement(s)
```

Notes:

- We can add `else` and `elif` statements to the inner `if` statement as required.
- We can also insert inner `if` statement inside the outer `else` or `elif` statements (if they exist).
- We can nest multiple layers of if statements.

Conditional Statements in bash

An example shell script:

```
1  #!/bin/bash
2
3  if [ $1 -gt 100 ]
4  then
5    echo Hey that\'s a large number.
6    pwd
7  fi
8
9  date
```

Line 3: Check if the first command line argument is greater than 100.
Line 5 and 6: Will only run if the test on line 3 returns `True`.
Line 5: The backslash escapes the special meaning of the single quote.
Line 7: `fi` signals the end of the `if` statement. All commands after this will be run as normal.
Line 9: Because this command is outside the `if` statement it will be run regardless of the outcome of the if statement.

Conditional Statements in bash

An example shell script:

```
1  #!/bin/bash
2
3  if [ $1 -gt 100 ]
4  then
5    echo Hey that\'s a large number.
6    pwd
7  fi
8
9  date
```

We run it:

```
user@bash: ./if_example.sh 15
Mon 29 Jan 19:11:28 2025
user@bash: ./if_example.sh 150
Hey that's a large number.
/home/ryan/bin
Mon 29 Jan 19:11:28 2025
user@bash:
```

Conditional Statements in bash

We are not limited to one conditional per statement:

```
1  #!/bin/bash
2  # Nested if statements
3
4  if [ $1 -gt 100 ]
5  then
6    echo Hey that\'s a large number.
7
8    if (( $1 % 2 == 0 ))
9    then
10     echo And is also an even number.
11   fi
12 fi
```

Conditional Statements in bash

We are not limited to one conditional per statement:

```
1  #!/bin/bash
2  # Nested if statements
3
4  if [ $1 -gt 100 ]
5  then
6    echo Hey that\'s a large number.
7
8    if (( $1 % 2 == 0 ))
9    then
10     echo And is also an even number.
11   fi
12 fi
```

Line 4: If the first command line argument is > 100, perform the following.
Line 8: For checking an expression we use double brackets.
Line 10: Runs only if both `if` statements are `True`.

Conditional Statements in bash

In bash, we have two sets of conditional operators, depending on whether the variables are numbers or strings.

For numbers:
`-eq`, `-ne`, `-gt`, `-ge`, `-lt`, `-le`

For strings:
=, `!=`, <, > (ASCII order for the last two), `-z`, `-n`

Conditional Statements in bash

In bash, we have two sets of conditional operators, depending on whether the variables are numbers or strings.

For numbers:
`-eq`, `-ne`, `-gt`, `-ge`, `-lt`, `-le`

For strings:
=, `!=`, <, > (ASCII order for the last two), `-z`, `-n`

To execute if the condition is `False`, prepend with `!`:

```
if ! [[ -e source1.txt ]]
```

Conditional Statements in bash

In bash, we have two sets of conditional operators, depending on whether the variables are numbers or strings.

For numbers:
`-eq`, `-ne`, `-gt`, `-ge`, `-lt`, `-le`

For strings:
=, `!=`, <, > (ASCII order for the last two), `-z`, `-n`

To execute if the condition is `False`, prepend with `!`:

```
if ! [[ -e source1.txt ]]
```

You can combine expressions:

```
if [[ -e source1.txt ]] && ! [[ -e source2.txt ]]
```

Loops

Loops are control structures that allow you to **repeat a set of instructions** several times. Along with conditionals, they are two of the most important tools when it comes to programming.

The two primary loops found in most programming languages are the `while` and `for` loops.

Loops

Loops are control structures that allow you to **repeat a set of instructions** several times. Along with conditionals, they are two of the most important tools when it comes to programming.

The two primary loops found in most programming languages are the `while` and `for` loops.

Definition

A `while`-loop is repeats a specific block of code sequentially as long as a certain condition is met.

Loops

Loops are control structures that allow you to **repeat a set of instructions** several times. Along with conditionals, they are two of the most important tools when it comes to programming.

The two primary loops found in most programming languages are the `while` and `for` loops.

Definition

A `while`-loop is repeats a specific block of code sequentially as long as a certain condition is met.

Definition

A `for`-loop is a set off block of code that contains a temporary variable known as an iterator, and runs the block of code over and over for different specified values of that iterator.

Loops in bash

In the example below we use a WHILE loop to print the numbers 1 to 10:

```
1  #!/bin/bash
2  # Basic while loop
3
4  counter=1
5  while [ $counter -le 10 ]
6  do
7    echo $counter
8    ((counter++))
9  done
10 echo All done
```

Line 4: We'll initialize the variable `counter` with it's starting value.
Line 5: While the condition is `True`, carry out the following commands.
Line 8: We increase the value of counter by 1.
Line 9: We're at the bottom of the loop so go back to line 5. If the condition is `True` then execute the commands. Otherwise continue executing any commands following `done`.

Loops in bash

bash also has an UNTIL loop which is quite similar to the WHILE loop. The difference is that it will execute the commands within it until the condition becomes `True`.

The WHILE loop would be able to handle the same scenarios. Sometimes, however, it just makes it a little easier to read if we phrase it with UNTIL rather than WHILE.

```
1  #!/bin/bash
2  # Basic until loop
3
4  counter=1
5  until [ $counter -gt 10 ]
6  do
7    echo $counter
8    ((counter++))
9  done
10 echo All done
```

Loops in bash

In the example below we use a FOR loop to print the elements of a list:

```
1  #!/bin/bash
2  # Basic for loop
3
4  headers='ra dec mag'
5
6  for header in $headers
7  do
8    echo $header
9  done
10 echo All done
```

Line 4: Create a simple list containing column headers.
Line 6: For each item in the list `headers` assign the item to the variable `header` and do the following commands.
Line 8: `echo` the header.
Line 10: `echo` another command to show that the `bash` script continued execution as normal after all the items in the list were processed.

Loops in bash

With a FOR loop in `bash`, we can also process a series of numbers:

```
1  #!/bin/bash
2  # Basic range in for loop
3  for value in {1..10}
4  do
5    echo $value
6  done
7  echo All done
```

It is important when specifying a range like this that there are no spaces present between the curly brackets. If there are then it will not be seen as a range but as a list of items.

Loops in bash

Multiple loops can be **nested** in case of iterating over more than one value in your code. This often happens when dealing with two-dimensional arrays.

Loops in bash

Multiple loops can be **nested** in case of iterating over more than one value in your code. This often happens when dealing with two-dimensional arrays.

```
#!/bin/bash

outer=(1 2 3 4 5)
inner=(a b c d e)

for i in ${outer[@]}
do
    for j in ${inner[@]}
    do
        echo "$i$j"
    done
done
```

Loops in bash

Most of the time your loops should run fine, but there are cases in which one wants to alter them slightly. For doing so, `bash` has two statements: **break and continue**.

The **break statement** tells `bash` to leave the loop straight away. For exampe, copying files should stop if the free disk space get's below a certain level.

```
#!/bin/bash

# Make a backup set of files for value in $1/*
do
  used=$( df $1 | tail -1 | awk '{ print $5 }' | sed 's/%//' )
  if [ $used -gt 90 ]
  then
    echo Low disk space 1>&2
    break
  fi
  cp $value $1/backup/
done
```

Loops in bash

The **continue statement** tells bash to stop running through this iteration of the loop and begin the next iteration.
For example, we use the loop to process a series of files but if we don't have the read permission for a file we should skip it.

```
#!/bin/bash

# Make a backup set of files for value in $1/*
do
  if [ ! -r $value ]
  then
    echo $value not readable 1>&2
    continue
  fi
  cp $value $1/backup/
done
```

Data Input/Output

A file is a container in computer storage devices used for storing data.

When we want to read from or write to a file, we need to open it first. When we are done, it needs to be closed so that the resources that are tied with the file are freed.

Generally, a **file operation** takes place in the following order:

- Open a file
- Read or write (perform operation)
- Close the file

Data Input/Output in bash

To **create** a new file containing output, we use a single redirect token:

```
#!/bin/sh

date > date.txt
```

We can check the content of `date.txt` when we run the script a few times:

```
$ bash ./date.sh
$ cat date.txt
Tue Jan 30 11:59:22 AM -03 2025
$ bash ./date.sh
$ cat date.txt
Tue Jan 30 11:59:41 AM -03 2025
```

Data Input/Output in bash

To **append** data to a file, use the double redirect tokens:

```
#!/bin/sh

TZ=UTC
date >> date.txt
```

We can check the content of `date.txt` when we run the script a few times:

```
$ bash ./date.sh
$ bash ./date.sh
$ bash ./date.sh
$ cat date.txt
Tue Jan 30 11:59:25 AM -03 2025
Tue Jan 30 11:59:32 AM -03 2025
Tue Jan 30 11:59:42 AM -03 2025
```

Data Input/Output in bash

To **open a file for reading**, we use redirection. In the following example, the built-in echo command prints the results of the redirect:

```
#!/bin/sh

echo $( < include.sh )
```

We can also read a file line by line:

```
#!/bin/sh

input="/path/to/txt/file"
while IFS= read -r line
do
  echo "$line"
done < "$input"
```

Functions

So far, we have used shell commands like `grep`, `cat`...

When we use them, we start (call) a command-line utility. Such utilities exist to provide commonly used functionalities without requiring us to write code on our own that carries out this functionalities.

Let's take a closer look at this.

Functions

To write code that is easy to read and easy to maintain, we generally apply the concepts of **decomposition and abstraction**.

Functions

To write code that is easy to read and easy to maintain, we generally apply the concepts of **decomposition and abstraction**.

Take the **example** of a projector. We can use a projector as a black box: We know what it does (display image from a connected computer). We don't need to know how it works. We know the interfaces (input, output).

This is the idea of **abstraction**.

Functions

To write code that is easy to read and easy to maintain, we generally apply the concepts of **decomposition and abstraction**.

Take the **example** of a projector. We can use a projector as a black box: We know what it does (display image from a connected computer). We don't need to know how it works. We know the interfaces (input, output).

This is the idea of **abstraction**.

The task of "projecting an image" is not carried out by a single device, but by multiple ones: a computer connected to a projector (which itself is composed of many parts, such as lamp, lens, circuits...), a remote to start it and so on. All devices used work together to produce the image.

This is the idea of **decomposition**.

Functions

When writing code, these concepts mean: **divide code into functions, modules, classes**.

Each of those units should be **self-contained**. They are used to break up code into reusable part and generally keep code organized and coherent.

In this class, we will achieve decomposition with **functions**.

Functions

Every function is composed of two pieces: **a header and body**.
The **function header** defines the name of the function and its argument(s).
The **function body** specifies contains the code the function carries out.

In many programming languages, it will look similar to this:

```
# Define the function
def add_three(input_var):
    output_var = input_var + 3
    return output_var
```

Functions

Every function is composed of two pieces: **a header and body**.
The **function header** defines the name of the function and its argument(s).
The **function body** specifies contains the code the function carries out.

In many programming languages, it will look similar to this:

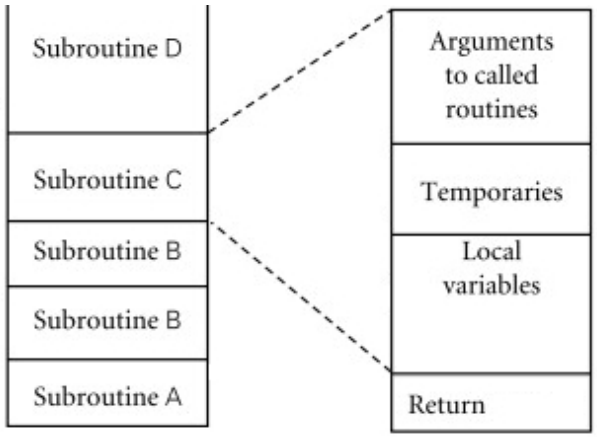
```
# Define the function
def add_three(input_var):
    output_var = input_var + 3
    return output_var
```

When we run a function, it is also referred to as **"calling" the function**.
In the code block below, we run the previously define function with 10 as the input value and assign the output to a new variable `new_number`.
In many programming languages, it will look similar to this:

```
# Run the function with 10 as input
new_number = add_three(10)
```

Subroutines

Modern programs are very large and complex. Programs more than a few dozen lines long are hard to read and understand **Subroutines** allow to break complex programs into simpler parts.
A subroutine takes input arguments and outputs a result - technically, they are functions. A subroutine can call further subroutines.



Example:

In a program that loads data, carries out computations based on them, and generates a plot, it would make sense to put these three parts into subroutines.

Subroutines and Functions in bash

When scripts get lengthy and/or repetitive, moving some parts into functions may be useful.
They can access the variables of the calling script (like exported variables in the shell). See the following example:

```
print_second_argument() {
if [[ $# -lt 2 ]] ; then
echo "$# argument given. At least two arguments are needed."
else
echo "The second argument is $2"
fi
}
```

The function is then called with:

```
print_second_argument $@
```

Summary

When talking about programming and script languages, it is useful to understand the basic terminologies:

Algorithm: A step-by-step procedure for solving a problem.

Syntax: The set of rules of a programming language.

Variable: A named storage location in memory that holds a value or data.

Data Type: A classification that specifies what type of data a variable can hold, such as integer, string, or boolean.

Operator: A symbol or keyword that represents an action or operation to be performed on one or more values or variables.

Statement: A single line or instruction in a program that performs a specific action or operation.

Function: A self-contained block of code that performs a specific task and can be called from other parts of the program.

Control Flow: The order in which statements are executed in a program, including loops and conditional statements.

Comment: Notes or explanations to the code that are ignored by the compiler or interpreter.

An Outlook: more complex algorithms

With now knowing about control flow statements, we can now write more complex code for calculations and file manipulation.

In this week's **tutorial session** we will write some more complex shell scripts for tasks common for working with astronomical data.

Once a code requires more **sophisticated analytical tools**, it becomes apparent that the built-in utilities as well as the ones we can write with too much effort are not sufficient.
In many programming languages, so-called **libraries** maintain functions that accomplish such complex tasks.

We will see how to use them later on when we learn how to write code in **Python**.