Python Basics

1. Introduction

2. Variables

3. Debugging

4. The first program

Introduction

Problem solving is a central part of computer science, the solutions that you create through the problem solving process are ***algorithms***. An algorithm is a step by step list of instructions that if followed exactly will solve the problem under consideration.

Problem solving is a central part of computer science, the solutions that you create through the problem solving process are ***algorithms***. An algorithm is a step by step list of instructions that if followed exactly will solve the problem under consideration.

For example, an algorithm to compute the length of a triangle’s hypotenuse might look like this:

Algorithm Example 1 (English):

- Ask for the lengths of the two sides
- Use Pythagorean Theorem to find hypotenuse
- Display the hypotenuse

Algorithms are like recipes: they must be followed exactly, they must be clear and unambiguous, and they must end. For improved precision, algorithms are often written in ***pseudocode***.

Algorithms are like recipes: they must be followed exactly, they must be clear and unambiguous, and they must end. For improved precision, algorithms are often written in ***pseudocode***.

Algorithm Example 2 (Pseudocode).

- Ask for length of non-hypotenuse sides:
    - Ask for first side's length. Call this side a.
    - Ask for second side's length. Call this side b.
- Let

$$hypotenuse = \sqrt{a^2 + b^2}$$

- Display the hypotenuse length

Once we have such a solution, we can use our computer to automate its execution. ***Programming*** is a skill that allows a computer scientist to take an algorithm and represent it in a notation (a program) that can be followed by a computer. A program is written in a ***programming language*** such as Python, the language you will learn in this course!

Once we have such a solution, we can use our computer to automate its execution. ***Programming*** is a skill that allows a computer scientist to take an algorithm and represent it in a notation (a program) that can be followed by a computer. A program is written in a ***programming language*** such as Python, the language you will learn in this course!

In [2]:

```
side_a = int(input("Enter the length of the first side:"))
side_b = int(input("Enter the length of the second side:"))
hypotenuse = (side_a**2 + side_b**2)**(1/2)
print("The hypotenuse of that triangle is:", hypotenuse)
```

```
Enter the length of the first side:3
Enter the length of the second side:4
The hypotenuse of that triangle is: 5.0
```

In [3]:
```
display_quiz(path+"algo.json", question_alignment='center', max_width=800)
```

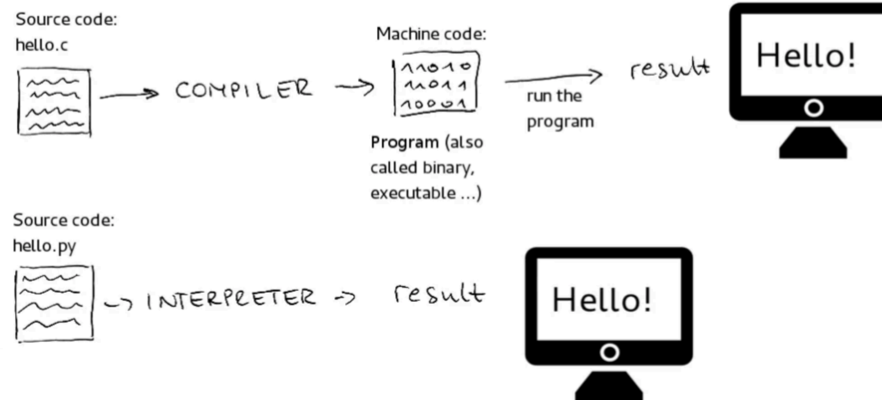
Which one is the definition of an algorithm?

- A special kind of notation used to describe how to solve a problem.
- It contains well-defined, unambiguous steps and must procude result in a finite time.
- It must be decribed using programming language.
- A step-by-step list of instructions for solving any instance of the problem that might arise

Hello, Python

Python emphasizes readability and is an ***interpreted language***, which for beginners means that Python instructions can be typed into an interactive prompt, or can be stored in a plain text file (called a `"script"`) and run later. These instructions are evaluated and the commands are executed by the Python ***interpreter***.

Python emphasizes readability and is an ***interpreted language***, which for beginners means that Python instructions can be typed into an interactive prompt, or can be stored in a plain text file (called a `"script"`) and run later. These instructions are evaluated and the commands are executed by the Python ***interpreter***.



A ***command***, often called a ***statement***, instructs the interpreter to do something. The first program described in many programming Language introductions is "Hello, World". This simple program demonstrates how a particular language produces a result, also how a language represents text and outputs a nominal greeting 🙂

A ***command***, often called a ***statement***, instructs the interpreter to do something. The first program described in many programming Language introductions is "Hello, World". This simple program demonstrates how a particular language produces a result, also how a language represents text and outputs a nominal greeting 🙂

In [4]:
```
print('Hello, World!')
```

```
Hello, World!
```

A ***command***, often called a ***statement***, instructs the interpreter to do something. The first program described in many programming Language introductions is "Hello, World". This simple program demonstrates how a particular language produces a result, also how a language represents text and outputs a nominal greeting 🙂

In [4]:
```
print('Hello, World!')
```

```
Hello, World!
```

There are several aspects to note even in this simple Python statement.

- First, `print()` is a built-in ***function***, a pre-defined operation that Python can use to produce *output*, a result of the program that will be made visible to the user. The print is followed by opening and closing parentheses; what comes between those parentheses is the value or ***arguments*** to be printed.

- Second, fixed values such as numbers, letters, and strings, are called ***constants*** which is a **data** where their value does not change. String constants use single quotes `'` or double quotes `"` in Python.

- Second, fixed values such as numbers, letters, and strings, are called ***constants*** which is a **data** where their value does not change. String constants use single quotes `'` or double quotes `"` in Python.

Notice that it is possible for statements to span more than one line using `\` or print multiple objects seperated by `,`.

- Second, fixed values such as numbers, letters, and strings, are called ***constants*** which is a **data** where their value does not change. String constants use single quotes `'` or double quotes `"` in Python.

Notice that it is possible for statements to span more than one line using `\` or print multiple objects seperated by `,`.

In [5]:
```
print('Hello,\
 World')

print('Hello', 'World')
```

```
Hello, World
Hello World
```

- Second, fixed values such as numbers, letters, and strings, are called ***constants*** which is a **data** where their value does not change. String constants use single quotes `'` or double quotes `"` in Python.

Notice that it is possible for statements to span more than one line using `\` or print multiple objects seperated by `,`.

In [5]:
```
print('Hello,\
 World')

print('Hello', 'World')
```

```
Hello, World
Hello World
```

In [6]:
```
'hi python'
'Hello, World!'
```

Out[6]:
```
'Hello, World!'
```

In [7]:

```
display_quiz(path+"print.json", max_width=800)
```

What appears in the output window when the following statement executes?

```
print("Hello!")
```

Nothing is printed. It generates a runtime error.	Hello!
哈囉!	"Hello!"

Using string methods like a word processor

One of the simplest tasks you can do with strings is to change the case of the words in a string.

One of the simplest tasks you can do with strings is to change the case of the words in a string.

In [8]:
```
print('hi python'.title())
```

```
Hi Python
```

One of the simplest tasks you can do with strings is to change the case of the words in a string.

In [8]:
```
print('hi python'.title())
```

```
Hi Python
```

In this example, we have the lowercase string 'hi python'. The ***method*** `title()` appears after the string in the `print()` call. A method is an action that Python can perform on a piece of data. The dot (`.`) after the string tells Python to make the `title()` method act on the string. Every method is followed by a set of parentheses that can accept arguments just like a function.

There are also other useful methods for string

There are also other useful methods for string

In [9]:
```
print('hi python'.upper())   # change a string to all uppercase
print('Hello World'.lower()) # change a string to all lowercase
print(' hi python '.strip()) # remove extra whitespace on the right and left .
```

```
HI PYTHON
hello world
hi python
```

There are also other useful methods for string

In [9]:
```
print('hi python'.upper())   # change a string to all uppercase
print('Hello World'.lower()) # change a string to all lowercase
print(' hi python '.strip()) # remove extra whitespace on the right and left .
```

```
HI PYTHON
hello world
hi python
```

These example statements introduce another language feature. The `#` symbol denotes the beginning of a ***comment***, a human-readable notation to the Python code that will be ignored by the computer when executed. A high-level description at the top of a script introduces a human reader to the overall purpose and methodology used in the script. All of the characters to the right of the `#` until the end of the line are ignored by Python.

> ## Exercise 1: Complete the following items to make sure you correctly set up the environment.

1. Open the explorer on the left-hand side or open the jupyter notebook.
2. Connect to the Python environment
3. Create a new code cell below and write a code snippet that prints out "finish". Execute the cell.
4. Create a new script called "finish.py" and write a code snippet that prints out "finish". Execute the script.

Exercise 1: Complete the following items to make sure you correctly set up the environment.

1. Open the explorer on the left-hand side or open the jupyter notebook.
2. Connect to the Python environment
3. Create a new code cell below and write a code snippet that prints out "finish". Execute the cell.
4. Create a new script called "finish.py" and write a code snippet that prints out "finish". Execute the script.

In [10]:
```
# Your code below
```

Exercise 1: Complete the following items to make sure you correctly set up the environment.

1. Open the explorer on the left-hand side or open the jupyter notebook.
2. Connect to the Python environment
3. Create a new code cell below and write a code snippet that prints out "finish". Execute the cell.
4. Create a new script called "finish.py" and write a code snippet that prints out "finish". Execute the script.

In [10]:
```
# Your code below
```

In [ ]:
```
%run finish.py
```

Operators and Expressions

Using operand like a calculator

Besides string, numbers are often used in programming. Python's built-in operators allow numeric values to be combined in a variety of familiar ways. Note that in Python, `2 + 3` is called an ***expression***, which consists of values/operands (such as `2` or `3`) and operators (such as `+`), and they are special statements!

Besides string, numbers are often used in programming. Python's built-in operators allow numeric values to be combined in a variety of familiar ways. Note that in Python, `2 + 3` is called an ***expression***, which consists of values/operands (such as `2` or `3`) and operators (such as `+`), and they are special statements!

In [12]:
```
# Integer
print(3+4)        # Prints “7”, which is 3 plus 4.
print(5-6)        # Prints “-1”, which is 5 minus 6
print(7*8)        # Prints “56”, which is 7 times 8
print(45/4)       # Prints “11.25”, which is 45 divided by 4, / is float(true)
print(2**10)      # Prints “1024”, which is 2 to the 10th power
```

```
7
-1
56
11.25
1024
```

When an operation such as forty-five divided by four produces a non-integer result, such as `11.25`, Python implicitly switches to a **floating-point** representation. When purely integer answers are desired, a different set of operators can be used.

When an operation such as forty-five divided by four produces a non-integer result, such as `11.25`, Python implicitly switches to a **floating-point** representation. When purely integer answers are desired, a different set of operators can be used.

In [13]:

```
print(45//4)     # Prints “11”, which is 45 integer divided by 4, // is floor
print(45%4)      # Prints “1”, because 4 * 11 + 1 = 45
```

```
11
1
```

When an operation such as forty-five divided by four produces a non-integer result, such as `11.25`, Python implicitly switches to a **floating-point** representation. When purely integer answers are desired, a different set of operators can be used.

In [13]:
```
print(45//4)      # Prints “11”, which is 45 integer divided by 4, // is floor
print(45%4)       # Prints “1”, because 4 * 11 + 1 = 45
```

```
11
1
```

The double slash signifies the integer ***floor division*** operator, while the percentage symbol signifies the modulus, or remainder operator.

In [14]:
```
display_quiz(path+"type.json", max_width=800)
```

What value is printed when the following statement executes?

```
print(18.0 // 4)
```

4	5
4.0	4.5

When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**. Python follows the same precedence rules for its mathematical operators that mathematics does.

When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**. Python follows the same precedence rules for its mathematical operators that mathematics does.

In [15]:
```
print(3-1*2)
print((3-1)*2)
print(2**4/2)
```

```
1
4
8.0
```

In [16]:
```
display_quiz(path+"precedence.json", max_width=800)
```

What is the value of the follwing expression?

```
16 - 2 * 5 // 3 + 1
```

14	24
13.667	3

String values also can be combined and manipulated in some intuitive ways.

String values also can be combined and manipulated in some intuitive ways.

In [17]:
```
s = 'hello' + 'world'
t = s * 4
print(s)
print(t)
```

```
helloworld
helloworldhelloworldhelloworldhelloworld
```

String values also can be combined and manipulated in some intuitive ways.

In [17]:
```
s = 'hello' + 'world'
t = s * 4
print(s)
print(t)
```

```
helloworld
helloworldhelloworldhelloworldhelloworld
```

The plus operator **concatenates** string values, while the multiplication operator **replicates** string values.

Variables

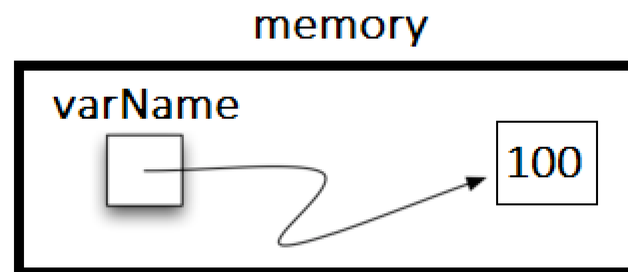
A ***variable*** is like a box in the computer's memory where you can store value. If you want to use the result of an evaluated expression later in your program, you can save it inside a variable!

A ***variable*** is like a box in the computer’s memory where you can store value. If you want to use the result of an evaluated expression later in your program, you can save it inside a variable!

You'll store values in variables with an ***assignment statement***. An assignment statement consists of a variable name, an equal sign, and the value to be stored. **In Python, every single thing is stored as an object. A Python variable is actually a reference to an object!**

A ***variable*** is like a box in the computer's memory where you can store value. If you want to use the result of an evaluated expression later in your program, you can save it inside a variable!

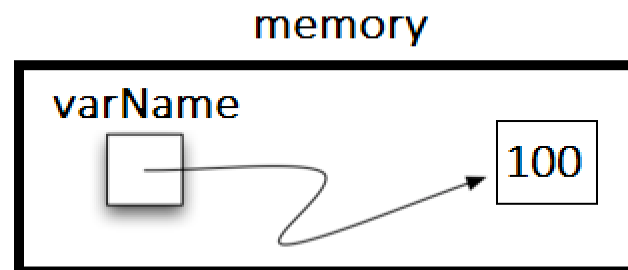
You'll store values in variables with an ***assignment statement***. An assignment statement consists of a variable name, an equal sign, and the value to be stored. **In Python, every single thing is stored as an object. A Python variable is actually a reference to an object!**



source: https://cs.berea.edu//cpp4python/AtomicData/AtomicData.html

A ***variable*** is like a box in the computer’s memory where you can store value. If you want to use the result of an evaluated expression later in your program, you can save it inside a variable!

You'll store values in variables with an ***assignment statement***. An assignment statement consists of a variable name, an equal sign, and the value to be stored. **In Python, every single thing is stored as an object. A Python variable is actually a reference to an object!**



source: https://cs.berea.edu//cpp4python/AtomicData/AtomicData.html

In [18]:
```
varName = 100
```

A variable is created the first time a value is stored in it. After that, you can use it in statements with other variables and values. When a variable is assigned a new value, the old value is forgotten. This is called **overwriting** the variable.

A variable is created the first time a value is stored in it. After that, you can use it in statements with other variables and values. When a variable is assigned a new value, the old value is forgotten. This is called **overwriting** the variable.

In [19]:
```
spam = 'Hello'   # 'Hello' is a string object
print(spam)      # spam is a variable, it is just a reference or tag
spam = 'Goodbye' # 'Goodbye' is another string object
print(spam)
```

```
Hello
Goodbye
```

In [20]:
```
display_quiz(path+"assignment.json", max_width=800)
```

What is printed when the following statements execute?

```
day = "Thursday"
day = 32.5
day = 19
print(day)
```

19	32.5
Thursday	Nothing is printed. A runtime error occurs.

The naming of variables is largely up to the user in Python. Python's simple rules are that variable names must begin with an alphabet letter or the underscore character, and may consist of an arbitrary number of letters, digits, and the underscore character (A-z, 0-9, and _).

The naming of variables is largely up to the user in Python. Python's simple rules are that variable names must begin with an alphabet letter or the underscore character, and may consist of an arbitrary number of letters, digits, and the underscore character (A-z, 0-9, and _).

Valid variable names	**Invalid variable names**
current_balance	current-balance (hyphens are not allowed)
currentBalance	current balance (spaces are not allowed)
account4	4account (can’t begin with a number)
_42	42 (can’t begin with a number)
TOTAL_SUM	TOTAL_*UM (special characters like * are not allowed)
hello	'hello' (special characters like ' are not allowed)

Python variable names are **case-sensitive**, meaning that capitalization matters. A variable named `size` is treated as distinct from variables named `Size` or `SIZE`.

Python variable names are **case-sensitive**, meaning that capitalization matters. A variable named `size` is treated as distinct from variables named `Size` or `SIZE`.

A small number of ***keywords***, names that are reserved for special meaning in Python, cannot be used as variable names. You can view this list by accessing the built-in Python help system.

Python variable names are **case-sensitive**, meaning that capitalization matters. A variable named `size` is treated as distinct from variables named `Size` or `SIZE`.

A small number of ***keywords***, names that are reserved for special meaning in Python, cannot be used as variable names. You can view this list by accessing the built-in Python help system.

In [21]:
```
help('keywords')
```

```
Here is a list of the Python keywords.  Enter any keyword to get more
help.

False               break               for                 not
None                class               from                or
True                continue            global              pass
__peg_parser__      def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
```

> Exercise 2: Ask AI tools to explain the rules for naming variables in Python.

- ChatGPT
- Gemini
- Copilot

Refer to https://hackmd.io/@phonchi/LLM_Tutor

Variables can be used to store all of the types of data values that Python is able to represent.

Variables can be used to store all of the types of data values that Python is able to represent.

In [22]:

```
my_string = 'characters'
my_Boolean = True # True/False
my_integer = 5
my_floating_point = 26.2
my_complex = 2+1j # Note that 1 can not be omitted

# You can condense the above statements into one line separated by ;
my_string = 'characters'; my_Boolean = True; my_integer = 5; my_floating_poin


### Multiple Assignment!
# You can also assign values to more than one variable using just a single li
my_string, my_Boolean, my_integer, my_floating_point, my_complex = 'character
```

In [23]:

```
print(10)
print(3.14)
print(2e10)   # scientific notation (https://en.wikipedia.org/wiki/Scientific_
print(12_000) # you can group digits using underscores to make large numbers r
print(3+2j)
```

```
10
3.14
20000000000.0
12000
(3+2j)
```

In [23]:
```
print(10)
print(3.14)
print(2e10)   # scientific notation (https://en.wikipedia.org/wiki/Scientific_
print(12_000) # you can group digits using underscores to make large numbers r
print(3+2j)
```

```
10
3.14
20000000000.0
12000
(3+2j)
```

Note that when you’re writing long numbers, you can group digits using underscores to make large numbers more readable. In addition, `print()` can be used to print any numerical number including those in scientific notation.

Data types

In Python variables and constants have a ***type***. We can ask Python what type something is by using the `type()` function

In Python variables and constants have a ***type***. We can ask Python what type something is by using the `type()` function

In [24]:
```
type('Hello, World!'), type(False), type(4), type(3.2), type(3+5j)
```

Out[24]:
```
(str, bool, int, float, complex)
```

In Python variables and constants have a ***type***. We can ask Python what type something is by using the `type()` function

In [24]:
```
type('Hello, World!'), type(False), type(4), type(3.2), type(3+5j)
```

Out[24]:
```
(str, bool, int, float, complex)
```

In [25]:
```
type(my_string),  type(my_Boolean), type(my_integer), type(my_floating_point)
```

Out[25]:
```
(str, bool, int, float, complex)
```

You can convert object of one type to another using ***cast*** by `str()`, `float()`, `int()`, etc.

You can convert object of one type to another using ***cast*** by `str()`, `float()`, `int()`, etc.

In [26]:
```
float(3)
```

Out[26]:
```
3.0
```

You can convert object of one type to another using ***cast*** by `str()`, `float()`, `int()`, etc.

In [26]:
```
float(3)
```

Out[26]:
```
3.0
```

In [27]:
```
int(3.9)
```

Out[27]:
```
3
```

You can convert object of one type to another using ***cast*** by `str()`, `float()`, `int()`, etc.

```
In [26]: float(3)
```

```
Out[26]: 3.0
```

```
In [27]: int(3.9)
```

```
Out[27]: 3
```

```
In [28]: int('3')
```

```
Out[28]: 3
```

You can convert object of one type to another using ***cast*** by `str()`, `float()`, `int()`, etc.

In [26]:
```
float(3)
```
Out[26]:
```
3.0
```
In [27]:
```
int(3.9)
```
Out[27]:
```
3
```
In [28]:
```
int('3')
```
Out[28]:
```
3
```
In [29]:
```
str(3)
```
Out[29]:
```
'3'
```

Python `ord()` and `chr()` are built-in functions. They are used to convert a character to an int and vice versa. Python `ord()` and `chr()` functions are exactly opposite of each other.

Python `ord()` function takes string argument of a single Unicode character and return its integer Unicode code point value. Let's look at some examples of using `ord()` function.

Python `ord()` and `chr()` are built-in functions. They are used to convert a character to an int and vice versa. Python `ord()` and `chr()` functions are exactly opposite of each other.

Python `ord()` function takes string argument of a single Unicode character and return its integer Unicode code point value. Let's look at some examples of using `ord()` function.

In [30]:
```
x = ord('A')
print(x)
```

```
65
```

Python `ord()` and `chr()` are built-in functions. They are used to convert a character to an int and vice versa. Python `ord()` and `chr()` functions are exactly opposite of each other.

Python `ord()` function takes string argument of a single Unicode character and return its integer Unicode code point value. Let's look at some examples of using `ord()` function.

In [30]:
```
x = ord('A')
print(x)
```

```
65
```

Python `chr()` function takes integer argument and return the string representing a character at that code point.

Python `ord()` and `chr()` are built-in functions. They are used to convert a character to an int and vice versa. Python `ord()` and `chr()` functions are exactly opposite of each other.

Python `ord()` function takes string argument of a single Unicode character and return its integer Unicode code point value. Let's look at some examples of using `ord()` function.

In [30]:
```
x = ord('A')
print(x)
```

```
65
```

Python `chr()` function takes integer argument and return the string representing a character at that code point.

In [31]:
```
y = chr(65)
print(y)
```

```
A
```

Conversion

There are two types of type conversion in Python.

- Implicit Conversion - automatic type conversion. Python always converts smaller data types to larger data types to avoid the loss of data.
- Explicit Conversion - manual type conversion

There are two types of type conversion in Python.

- Implicit Conversion - automatic type conversion. Python always converts smaller data types to larger data types to avoid the loss of data.
- Explicit Conversion - manual type conversion

In [32]:
```
5 + 4.2 # Implicit conversion
```

Out[32]:
```
9.2
```

There are two types of type conversion in Python.

- Implicit Conversion - automatic type conversion. Python always converts smaller data types to larger data types to avoid the loss of data.
- Explicit Conversion - manual type conversion

In [32]:
```
5 + 4.2 # Implicit conversion
```

Out[32]:
```
9.2
```

In Python, `complex > float > int > bool`

There are two types of type conversion in Python.

- Implicit Conversion - automatic type conversion. Python always converts smaller data types to larger data types to avoid the loss of data.
- Explicit Conversion - manual type conversion

```
In [32]: 5 + 4.2 # Implicit conversion
```

```
Out[32]: 9.2
```

In Python, `complex > float > int > bool`

```
In [33]: int(4.7) + 3 # Explicit conversion
```

```
Out[33]: 7
```

In [34]:

```
display_quiz(path+"ex_type.json", max_width=800)
```

What value is printed when the following statement executes?

```
print(int(53.785))
```

- 53.785
- 54
- Nothing is printed. It generates a runtime error.
- 53

Debugging

Programming languages are not very forgiving for beginners, and a great deal of time learning to write software can be spent trying to find bugs, or errors in the code. Locating such bugs and correcting them is thus known as ***debugging***.

Programming languages are not very forgiving for beginners, and a great deal of time learning to write software can be spent trying to find bugs, or errors in the code. Locating such bugs and correcting them is thus known as ***debugging***.

There are three major classes of bug that we create in software:

- ***syntax errors*** (mistakes in the symbols that have been typed)
- ***semantic errors*** (mistakes in the meaning of the program)
- ***runtime errors*** (mistakes that occur when the program is executed.)

Syntax errors are the most common for novices, and include simple errors such as forgetting one of the quote marks at the beginning or ending of a text string, failing to close open parentheses, or misspelling the function name `print()`. As examples:

Syntax errors are the most common for novices, and include simple errors such as forgetting one of the quote marks at the beginning or ending of a text string, failing to close open parentheses, or misspelling the function name `print()`. As examples:

In [35]:
```
print(5 + )
```

```
  File "C:\Users\adm\AppData\Local\Temp\ipykernel_61744\2298961889.p
y", line 1
    print(5 + )
              ^
SyntaxError: invalid syntax
```

Syntax errors are the most common for novices, and include simple errors such as forgetting one of the quote marks at the beginning or ending of a text string, failing to close open parentheses, or misspelling the function name `print()`. As examples:

In [35]:
```
print(5 + )
```

```
  File "C:\Users\adm\AppData\Local\Temp\ipykernel_61744\2298961889.p
y", line 1
    print(5 + )
              ^
SyntaxError: invalid syntax
```

This expression is missing a value between the addition operator and the closing parenthesis.

In [36]:

```
print(mystring)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_61744\551408598.py in <module>
----> 1 print(mystring)

NameError: name 'mystring' is not defined
```

In [36]:
```
print(mystring)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_61744\551408598.py in <module>
----> 1 print(mystring)

NameError: name 'mystring' is not defined
```

In this case it found a name error and reports that the variable being printed has not been defined. Python can't identify the variable name provided.

In [37]:

```
pront(5)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_61744\1320233283.py in <module>
----> 1 pront(5)

NameError: name 'pront' is not defined
```

In [37]:

```
pront(5)
```

```
---------------------------------------------------------------------------
-----
NameError                                 Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_61744\1320233283.py in <module>
----> 1 pront(5)

NameError: name 'pront' is not defined
```

Like calling someone by the wrong name, misspelling the name of a known function or variable can result in confusion and embarrassment.

Semantic errors are flaws in the algorithm, or flaws in the way the algorithm is expressed in a language. Examples might include using the wrong variable name data type in a calculation, or getting the order of arithmetic operations wrong in a complex expression.

Semantic errors are flaws in the algorithm, or flaws in the way the algorithm is expressed in a language. Examples might include using the wrong variable name data type in a calculation, or getting the order of arithmetic operations wrong in a complex expression.

In [38]:
```
num1 = input('Enter a number:')
num2 = input('Enter another number:')
sum_var = num1 + num2

print('The sum of', num1, 'and', num2, 'is', sum_var)
```

```
Enter a number:3
Enter another number:2
The sum of 3 and 2 is 32
```

Semantic errors are flaws in the algorithm, or flaws in the way the algorithm is expressed in a language. Examples might include using the wrong variable name data type in a calculation, or getting the order of arithmetic operations wrong in a complex expression.

In [38]:
```
num1 = input('Enter a number:')
num2 = input('Enter another number:')
sum_var = num1 + num2

print('The sum of', num1, 'and', num2, 'is', sum_var)
```

```
Enter a number:3
Enter another number:2
The sum of 3 and 2 is 32
```

The error is that the program performs concatenation instead of addition, because the programmer failed to write the code necessary to convert the inputs to integers.

Semantic errors are flaws in the algorithm, or flaws in the way the algorithm is expressed in a language. Examples might include using the wrong variable name data type in a calculation, or getting the order of arithmetic operations wrong in a complex expression.

In [38]:
```
num1 = input('Enter a number:')
num2 = input('Enter another number:')
sum_var = num1 + num2

print('The sum of', num1, 'and', num2, 'is', sum_var)
```

```
Enter a number:3
Enter another number:2
The sum of 3 and 2 is 32
```

The error is that the program performs concatenation instead of addition, because the programmer failed to write the code necessary to convert the inputs to integers.

The `input()` function waits for the user to type some text on the keyboard and press ENTER and **returns a string value**. It allows the programmer to provide a prompt string.

Finally, runtime errors at this level might include unintentionally dividing by zero or using a variable before you have defined it. Python reads statements from top to bottom, and it must see an assignment statement to a variable before that variable is used in an expression.

Finally, runtime errors at this level might include unintentionally dividing by zero or using a variable before you have defined it. Python reads statements from top to bottom, and it must see an assignment statement to a variable before that variable is used in an expression.

In [39]:
```
5/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_61744\2874912419.py in <module>
----> 1 5/0

ZeroDivisionError: division by zero
```

In [40]:
```
display_quiz(path+"error.json", max_width=800)
```

Which of the following is a semantic error?

- Attempting to divide by 0.
- Forgetting the closing parenthesis) on a print statement.
- Forgetting to divide by 100 when printing a percentage amount.

Exercise 3: Employ AI tools to diagnose errors.

The first program

While the interactive shell is good for running Python instructions one at a time, sometimes you have to use a ***script***, to write entire Python programs. In this case, you'll type the instructions into the file editor.

While the interactive shell is good for running Python instructions one at a time, sometimes you have to use a ***script***, to write entire Python programs. In this case, you'll type the instructions into the file editor.

In [41]:
```
%%writefile hello.py
"""
This program says hello and asks for your name.
It also ask the age of you.
"""

print('Hello, world!')
myName = input('What is your name? ') # ask for their name
print('It is good to meet you, ' + myName)
print('The length of your name is:\n' + str(len(myName)))
myAge = input('What is your age? ') # ask for their age
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

```
Overwriting hello.py
```

Once you’ve entered your source code, the ipython **magic** `%%writefile` will save it so that you won’t have to retype it each time you start. You can then use another magic `%run` to execute the python script.

Once you've entered your source code, the ipython **magic** `%%writefile` will save it so that you won't have to retype it each time you start. You can then use another magic `%run` to execute the python script.

In [ ]:
```
%run hello.py
```

Exercise 4: Utilize AI tools to explain the program or add comments to the program.

> Exercise 5: Write a script that inputs a five-digit integer from the user. Separate the number into its individual digits. Print them separated by three spaces each. For example, if the user types in the number `42339`, the script should print `4   2   3   3   9`

Hint: Use floor division (`//`) and remainder (`%`) to isolate the digits.

In [ ]:

```
# Your answer here
# x=42339
# Get the user's input from their keyboard and convert it to integer:
x = _____('Enter a 5 digit integer')
# Get the last digit by remainder
digits4 =
# Perform floor division and get the remainig digits
x =
#....

# Print out the results
____(digits0,'   ',digits1,'   ',digits2,'   ',digits3,'   ',digits4)
```

Exercise 5: Employ AI tools to write a program or enhance the program.

In [43]:

```python
from jupytercards import display_flashcards
fpath= "https://raw.githubusercontent.com/phonchi/nsysu-math106A/refs/heads/m
display_flashcards(fpath + 'ch1.json')
```

algorithm

Next

>