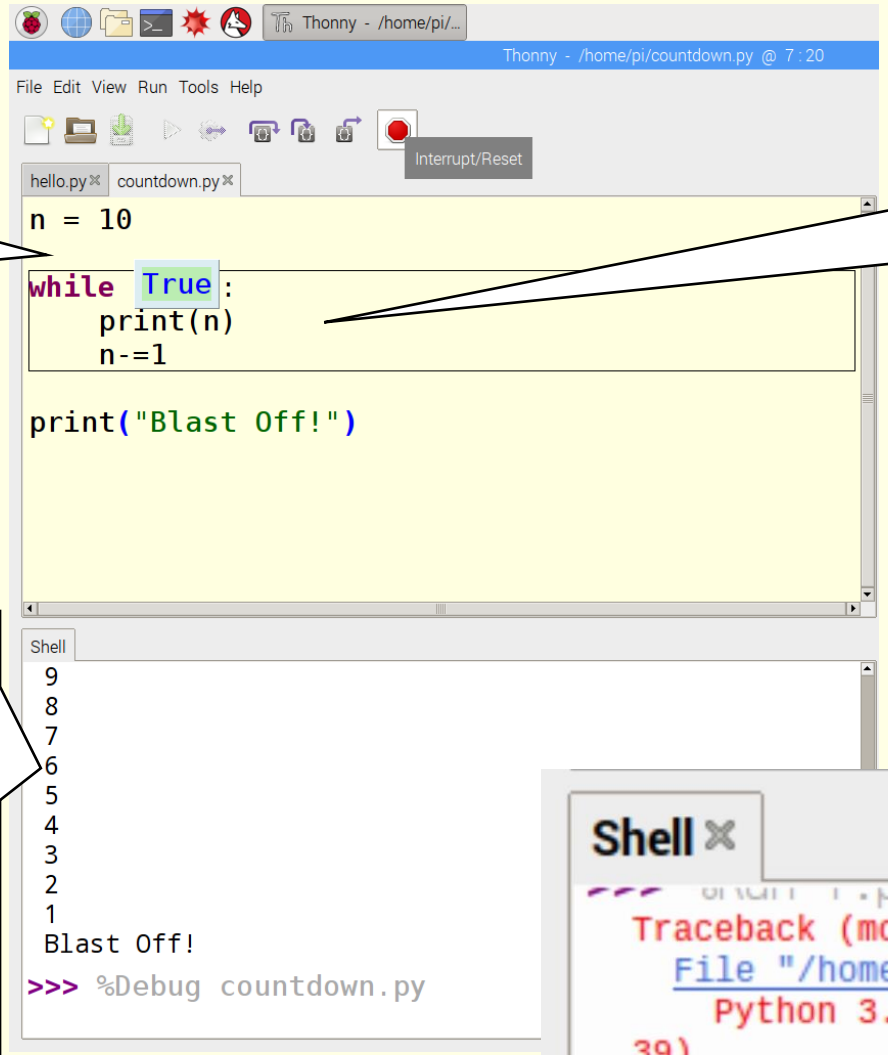
Learning Objectives

- Be familiar with the use of an IDE to develop and debug a program
- Understand the purpose of testing and devise a test plan
- Discuss the purpose of programming standards

Syntax Highlighting **Editor** **Run-time environment** **Debugger**

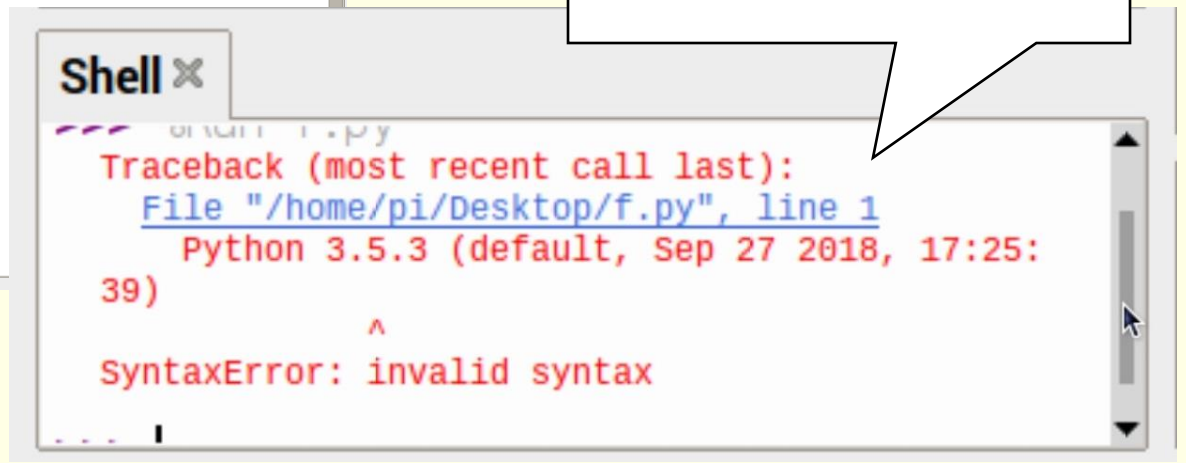


..... to enable program code to be entered / edited

.....to help make the code more readable and easier to maintain

..... to enable program to be run / to check for runtime errors / test the program

.....to display information about errors / location of errors / suggest solutions



The IDE

- **Software used to develop/debug a program.**
- At the very least an IDE should include:
  - An editor for writing **Source Code**
  - Facilities for automating the **Build**
  - A **Debugger**
  - **Breakpoints and steppers**
  - Features to help with code writing, such as **code completion.**

Key features

- **Code editors:** A text area used for programmers to enter code directly into the IDE. Often comes with additional features such as syntax highlighting, autocomplete code and self-indentation support.

- **Error diagnostics:** A feature which reports errors (mainly syntax) in the code and potential problems along with where they are found and often suggestions on how to fix.

- **Run-time environments:** Software to support the execution and running of programs. It allows the developer to run their code during development in order to check for logical errors.

- **Translators:** A program which converts high-level or assembly-level code into executable machine code by either compiling it or interpreting it.

- **Auto-documentation:** A feature which tracks variables declared by the programmer, modules and other special programmer comments in order to produce self-documentation to help program maintenance, debugging and support.

There are many other useful features depending on the IDE you are using…

- Underlines syntax errors dynamically
- Watch window
- Break points
- Error message list
- Step-mode
- Traces
- Crash-dump/post-mortem routine
- Stack contents
- Cross-referencers

Explained…

- **Underlines syntax errors dynamically** - can be corrected before running // saves times
- **Watch window -** View how variables change during running of the program
- **Break points -** Stop the program at set points to check the values of variables
- **Error message list -** Tells you where errors are and suggests corrections
- **Step-mode** (stepper) - Executes program one statement at a time to watch variable values and program pathways
- **Program Tracing -** Printouts of variable values for each statement execution within a program
- **Crash-dump/post-mortem routine** Shows the state of variables where an error occurs
- **Stack contents -** Shows sequencing through procedures/modules/ recursive algorithms
- **Cross-referencers** - Identifies where variables/constants are used in a program to avoid duplications

Test Data

Erroneous data

This type of data **should not be accepted** by the program or it will cause an error.

For example, an integer is entered when a string is required.

Boundary data

This type of data **should be accepted** by the program and is **valid**.

It is used to check that values entered at the **boundary** of a range will be accepted.

It also checks that data **just outside** the range is handled by the program.

Normal data

This type of data **should be accepted** by the program and is **valid**.

It is the normal data that you would expect to be entered into your program.

Worked Example

The following algorithm is intended to calculate and print the average mark for each student in a class, for all the tests they have attempted:

```
// average mark
students = input("How many students? ")

for n = 1 to students
   name = input("Enter student name ")
   totalMarks = input("Enter total marks for ", name)
   numTests =  input("How many tests has this student taken? ")
   averageMark = round(totalMarks/numTests)
   print ("Average mark = ",averageMark)
next n
```

The test plan will look something like this:

Test number	Test data	Purpose of test	Expected result	Actual result
1	Number of students = 4 for tests 1-4 Jo: total marks 27, tests 3	Normal data, integer result	9	9
2	Tom: total marks 31, tests 4	Normal data, non-integer result rounded up	8	8
3	Beth: total marks 28, tests 3	Normal data, result rounded down	9	9
4	Amina: total marks 0, tests 0	No tests taken	0	Program crashes
5	Number of students abc	Test invalid data	Program terminates	

You can probably think of some other input data that would make the program crash.
For example, what if the user enters 31.5 for the total marks? The program should validate all user input, so some amendments will have to be made to the program before general release!

Why have standards?

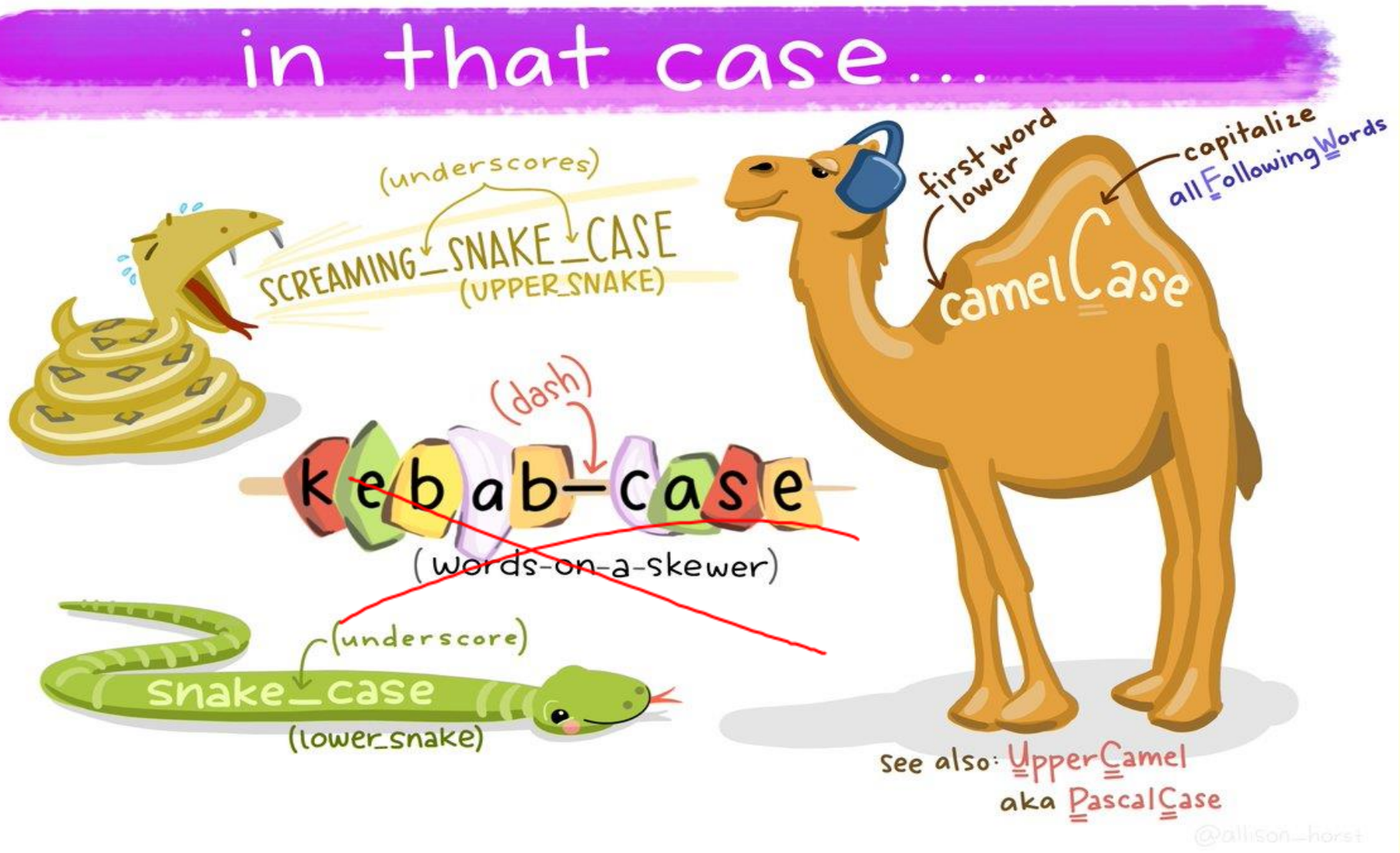
- When working with many team members and multiple projects in an organisation, using a coding standards policy is recommended
- Reduce complexity and aid readability
- This reduces complexity and makes the search for any bugs more straightforward.
- Reduce the errors.
- Easy to maintain
- Cost-efficient

Standards

Guidelines could include:

- You should always try to use meaningful names for variables, rather than x, y and z, as this helps to make the program easy to follow and update when required.
- It is also helpful, within a team of programmers, to have standards for naming variables and constants, as this will leave less room for errors and inconsistencies in the names in a large program.
- Start all variable names with a lowercase letter
- Do not use underscores in the middle of variable names
- Use "camelCaps" to separate parts of a variable name – for example, timeInMinutes, maxTemperature
- Do not use overly long names but keep them meaningful – maxTemp is better than maximumTemperature if there is not likely to be any confusion over the meaning of max
- Use all UPPERCASE letters for constants, which are then instantly identifiable
- When defining a class in object-oriented programming, start with an uppercase letter, with the rest of the class name lowercase
- Variables must not be set up outside the scope of a function: **this sets a limit on where to look for bugs and reduces the likelihood of a problem spread across many modules.**

in that case...


(underscores)
SCREAMING_SNAKE_CASE
(UPPER_SNAKE)
(dash)
kebab-case
(words-on-a-skewer)
(underscore)
snake_case
(lower_snake)
first word lower
capitalize all Following Words
camelCase
See also: UpperCamel aka PascalCase
@allison_horst

Function rules

- No function may be longer than a single
  complexity and aid readability.
- Comment your code - Do not do line-by
  look almost unreadable
- Indentation to aid readability
- Follow the single-entry/single-exit rule.
- Never write multiple return statements in the san

```
if (condition)
   return 42;
else
   return 97;
```

"This is ugly, you have to use a local variable!"

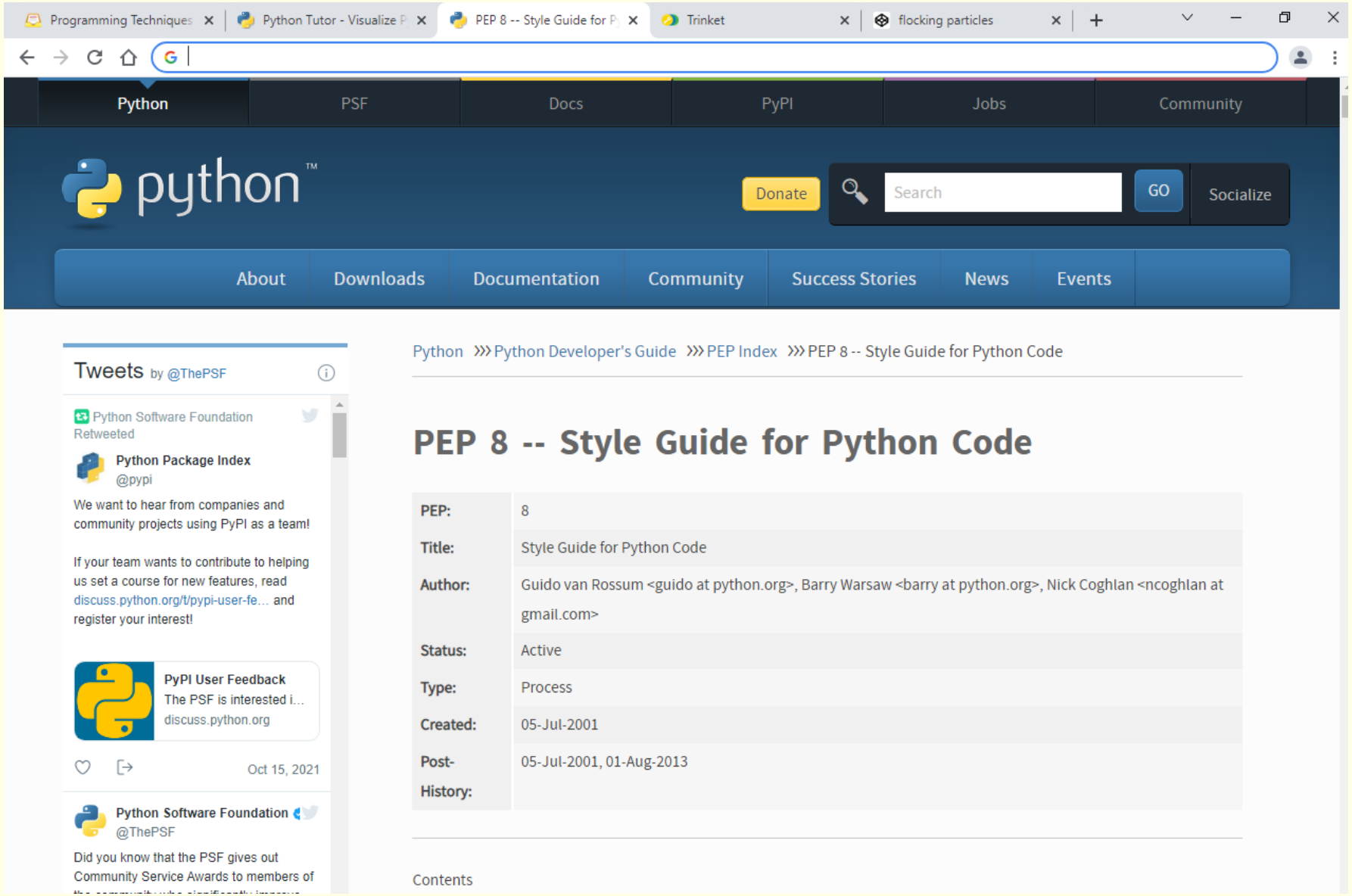
```
int result;
if (condition)
   result = 42;
else
   result = 97;
return result;
```

The main reason multiple exit points are bad is that they complicate control flow.

The more complicated the control flow is, the harder the code is to understand.

The harder the code is to understand, the greater the change of introduction bugs whenever the code is modified.

Official Python programming standards (PEP 8)



Programming Standards Summary

- Code Comment and Documentation
- Use of Indentation
- Commenting on Obvious Things
- Grouping Code
- Proper and Consistent Scheme for Naming
- CamelCase & UnderScore
- Deep nesting structure should be avoided