

# Welcome to Transcript's documentation!

Can't find what you're looking for? The [on-line version of this documentation](#) is frequently updated, reflecting the newest features and troubleshooting procedures.

## Table of Contents:

- 1. Transcript: what and why
  - 1.1. What is Transcript
  - 1.2. Language preferences, a subjective account
  - 1.3. The ecosystem: different batteries
  - 1.4. Code structure
  - 1.5. Debuggability
- 2. Getting started
  - 2.1. Installation
    - 2.1.1. Installation troubleshooting checklist
  - 2.2. Your first Transcript program
  - 2.3. Available command line switches
  - 2.4. Compiling to JavaScript 6
  - 2.5. Compiling for node.js
  - 2.6. Using sourcemaps and annotated target code
    - 2.6.1. Sourcemaps
    - 2.6.2. Annotated target code
  - 2.7. Static type validation
  - 2.8. Getting help and giving feedback
- 3. Special facilities
  - 3.1. Transcript's module mechanism
  - 3.2. Using browser stubs to test non-GUI code that uses `console.log` and `window.alert`
  - 3.3. Creating JavaScript objects with `__new__` (<constructor call>)

- 3.4. The `__pragma__` mechanism
  - 3.4.1. The function-like variety
  - 3.4.2. The comment-like variety
  - 3.4.3. The single-line activation variety
  - 3.4.4. The single-line deactivation variety
- 3.5. Identifier aliasing: `__pragma__` ('alias', ...) and `__pragma__` ('noalias', ...)
- 3.6. Generating `__doc__` attributes from docstrings: `__pragma__` ('docat') and `__pragma__` ('nodocat')
- 3.7. Skipping Transcrypt code fragments when running with CPython: `__pragma__` ('ecom') and `__pragma__` ('noecom')
- 3.8. Surpassing the speed of native JavaScript: `__pragma__` ('fcall') and `__pragma__` ('nofcall')
- 3.9. Enabling Python's *send* syntax: `__pragma__` ('gsend') and `__pragma__` ('nogsend')
- 3.10. Automatic conversion to iterable: `__pragma__` ('iconv') and `__pragma__` ('noiconv')
- 3.11. Conditional compilation: `__pragma__` ('ifdef', <symbol>), `__pragma__` ('ifndef', <symbol>), `__pragma__` ('else') and `__pragma__` ('endif')
- 3.12. Inserting literal JavaScript: `__pragma__` ('js', ...) and `__include__` (...)
- 3.13. `__pragma__` ('jskeys') and `__pragma__` ('nojskeys')
- 3.14. Keeping your code lean: `__pragma__` ('jsmod') and `__pragma__` ('nojsmod')
- 3.15. `__pragma__` ('keycheck') and `__pragma__` ('nokeycheck')
- 3.16. Keeping your code lean: `__pragma__` ('kwargs') and `__pragma__` ('nokwargs')
- 3.17. Preventing target annotation: `__pragma__` ('noanno')
- 3.18. Operator overloading: `__pragma__` ('opov') and `__pragma__` ('noopov')

- 3.19. Skipping fragments while generating code: `__pragma__` ('skip') and `__pragma__` ('noskip')
- 3.20. Automatic conversion to truth value: `__pragma__` ('tconv') and `__pragma__` ('notconv')
- 3.21. Adding directories to the module search path: `__pragma__` ('xpath', <directory list>)
- 3.22. Using an external transpiler: `__pragma__` ('xtrans', <translator>, ..., cwd = <workingdirectory>)
- 3.23. Create bare JavaScript objects and iterate over their attributes from Python: `__pragma__` ('jsiter') and `__pragma__` ('nojsiter')
- 4. Systematic code examples: a guided tour of Transcrypt
  - 4.1. Arguments: `**kwargs`, `*args`, defaults, at call and def time, also for lambda's
  - 4.2. Attribute access by name: `getattr`, `setattr`, `hasattr`
  - 4.3. Attribute proxies by name: `__getattr__`, `__setattr__`
  - 4.4. Bytes and bytearrays: initial support
  - 4.5. Callable or not: using the callable () built-in function
  - 4.6. Classes: multiple inheritance and assignment of bound functions
  - 4.7. Complex numbers: Python's builtin complex datatype
  - 4.8. Conditional expressions: simple and nested
  - 4.9. Control structures: `for...else`, `while...else`, `if...elif...else`, `break`, `continue`
  - 4.10. Data classes: Avoiding boilerplate code
  - 4.11. Data structures: tuple, list, dict, set
  - 4.12. Decorators: function and class, with and without parameters
  - 4.13. Dict comprehensions
  - 4.14. Dictionaries: dict revisited
  - 4.15. Diverse issues
  - 4.16. Diverse pulls
  - 4.17. Docstrings: `__doc__` attribute generated optionally
  - 4.18. Exceptions: exception class hierarchy, finally
  - 4.19. Extended slices: facilitating NumScript and such

- 4.20. General functions: sort and sorted
- 4.21. Global variable access by using globals ()  
[<variable\_name>]
- 4.22. Indices and slices: LHS, RHS, basic and extended
- 4.23. Iterators and generators
- 4.24. Lambda functions with all types of args
- 4.25. List comprehensions: multi-loop and nested with multiple if's
- 4.26. Local classes: inside other classes and functions
- 4.27. Metaclasses: overriding type.\_\_new\_\_ in a descendant metaclass
- 4.28. Method and class decorators
- 4.29. Module builtin: a small part of it demo'ed
- 4.30. Module cmath: almost all of Python's cmath module
- 4.31. Module datetime: transcription of Python's datetime module
- 4.32. Module itertools: almost all of Python's itertools module
- 4.33. Module math: almost all of Python's math module
- 4.34. Module random: most important functions of Python's random module
- 4.35. Module re: transcription of Python's re module
- 4.36. Module time: transcription of Python's time module
- 4.37. Modules: hierarchical, both local to the project and global url-based
- 4.38. Nonlocals
- 4.39. Operator overloading
- 4.40. Properties
- 4.41. Representation as text: the repr and str built-in functions
- 4.42. Set comprehensions
- 4.43. Super
- 4.44. Simple and augmented assignment

- 4.45. Truthyness: optional Python-style evaluation of truthyness, falsyness and non-empty container selection
  - 4.46. Tuple assignment: recursive and in for-headers using enumerate
- 5. Seamless interoperation with the DOM
  - 5.1. Practical example: a simple, responsive website using no HTML or CSS at all
  - 5.2. SVG example: Turtle graphics
- 6. Mixed examples
  - 6.1. Example: Pong
    - 6.1.1. Three ways of integration with JavaScript libraries
    - 6.1.2. Minification
  - 6.2. Example: jQuery
  - 6.3. Example: iOS web app with native look and feel
  - 6.4. Example: D3.js
  - 6.5. Example: React
  - 6.6. Example: Riot
  - 6.7. Example: Using input and print in a DOM `__terminal__` element in your browser
  - 6.8. Example: Using the Parcel.js bundler to package a set of modules written in diverse programming languages
- 7. Autotesting Transcrypt code
  - 7.1. Why it's needed
  - 7.2. How it works
- 8. The philosophy behind Transcrypt and its impact on design decisions
  - 8.1. Transcrypt's primary target audience
    - 8.1.1. Seasoned Python developers
    - 8.1.2. Seasoned JavaScript developers
    - 8.1.3. Other developers
  - 8.2. How to best serve this target audience
  - 8.3. Specific design choices made for Transcrypt and their underlying motivation

- 8.3.1. Why is Transcrypt written in Python and not in JavaScript
  - 8.3.2. Why does Transcrypt blend Python datatypes with JavaScript datatypes
  - 8.3.3. Why are certain Python constructions supported as a local (or global) option rather than by default
  - 8.3.4. Why were the `__pragma__`'s added
- 9. The main differences with CPython
  - 9.1. Differences due to the compiled, rather than interpreted nature of Transcrypt
  - 9.2. Differences due to the 'lean and mean' design goal
  - 9.3. Differences due to interoperability with JavaScript and JavaScript libraries
  - 9.4. Differences due to running Transcrypt applications in the browser, rather than on the desktop