- Remember to register your attendance using the UoB Check-In app. Either

1. download, install, and use the native app[a] available for Android and iOS, or
2. directly use the web-based app available at

https://check-in.bristol.ac.uk

noting the latter is also linked to via the `Attendance` menu item on the left-hand side of the Blackboard-based unit web-site.

- The hardware *and* software resources located in the MVB Linux lab(s). (e.g., MVB-1.15 or MVB-2.11) are managed by the Faculty IT Support Team, a subset of IT Services. If you encounter a problem (e.g., a workstation that fails to boot, an error when you try to use some software, or you just cannot log into your account), they can help: you can contact them, to report then resolve said problem, via

https://www.bristol.ac.uk/it-support

- The lab. worksheet is written *assuming* you work in the lab. using UoB-managed and thus *supported* equipment. If you need or prefer to use your own equipment, however, various *unsupported*[b] alternatives available: for example, *you* could 1) manually install any software dependencies yourself, *or* 2) use the unit-specific Vagrant[c] box by following instructions at

https://cs-uob.github.io/COMS30048/vm

- The purpose of the worksheet is to provide a) a tutorial-style introduction to selected technologies or concepts, and/or b) a means to explore them via hands-on tasks and challenges. Note that the worksheet is not assessed *at all*: if you are confident that you already understand the content, there is no problem with nor penalty for totally ignoring it.
- Keep in mind that various mechanisms exist to get support with and/or feedback on your work; these include both in-person (e.g., the lab. slot itself) *and* online (e.g., the unit forum, accessible via the unit web-site) instances.

[a] https://www.bristol.ac.uk/students/support/it/software-and-online-resources/registering-attendance

[b] The implication here is that such alternatives are provided in a best-effort attempt to help you: they are experimental, and so *no* guarantees about nor support for their use will be offered.

[c] https://www.vagrantup.com

COMS30048 lab. worksheet #2

Before you start work, download (and, if need be, unarchive[a]) the file

`https://assets.phoo.org/COMS30048_2025_TB-2/csdsp/sheet/lab-02.tar.gz`

somewhere secure[b] in your file system; from here on, we assume `${ARCHIVE}` denotes a path to the resulting, unarchived content. The archive content is intended to act as a starting point for your work, and will be referred to in what follows.

[a] For example, you could 1) use `tar`, e.g., by issuing the command `tar xvfz lab-02.tar.gz` in a terminal window, 2) use `ark` directly: use the `Activities` desktop menu item, search for and execute `ark`, use the `Archive`→`Open` menu item to open `lab-02.tar.gz`, then extract the contents via the `Extract` button, or 3) use `ark` indirectly: use the `Activities` desktop menu item, search for and execute `dolphin`, right-click on `lab-02.tar.gz`, select `Open with`, select `ark`, then extract the contents via the `Extract` button.

[b] For example, the `Private` sub-directory within your home directory (which, by default, cannot be read by another user).

1. Introduction

The AES[1] block cipher represents an important, standardised component within a wide range of use-cases. As a result, cryptographic software libraries will typically include an AES implementation[2] that supports multiple parameter sets (e.g., cipher key size) and/or optimisation goals (e.g., with respect to time or space) stemming from flexibility afforded by the underlying design. On one hand, using such an implementation as is, i.e., as a black-box, and so ignoring the internal, implementation detail, can be preferable and indeed advantageous. On the other hand, however, doing so is not *always* possible: some example scenarios include where

- an existing implementation is not available for a given platform, e.g., a particular (perhaps niche) micro-processor, or
- functional or behavioural properties of an existing implementation mean it is not fit for purpose, e.g., is not efficient enough (in time or space), or lacks countermeasures for a given attack technique.

Even beyond these examples, engaging with the implementation detail can also provide general insight into the design. The selection[3] of the Rijndael design as the AES standard was motivated, in part, by practically-oriented criteria such as efficiency; design strategies in Rijndael that help satisfy such criteria, while *also* guaranteeing resistance against cryptanalysis, are arguably made clearer by studying implementations of it.

Accepting the above premise as true, the goal of this lab. worksheet is to 1) improve your understanding of AES from a theoretical perspective, so focusing on the underlying design, then 2) switch to a more practical focus by developing an AES implementation of your own (in a programming language of your choice).

2. AES in theory: improve your understanding

AES is a fairly "clean" design, which combines 1) simple high-level structure based on an SP-network[4], with 2) a close connection to underlying Mathematics at lower-levels. The lecture slot(s) already provide a fairly comprehensive overview, but, as an alternative or supplement, you could consider use of other resources such as

- the FIPS-197 [1] standard

  `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf`

- relevant video-based overviews, such as

  `http://www.youtube.com/channel/UC1usFRN4LCMcfIV7UjHNuQg`

  that includes block cipher material in

  `http://www.youtube.com/watch?v=x1v2tX4_dkQ`
  `http://www.youtube.com/watch?v=NHuibtoL_qk`
  `http://www.youtube.com/watch?v=4FBgb2uobWI`

  and
- less formal introductions such as

  `http://www.moserware.com/2009/09/stick-figure-guide-to-advanced.html`

[1] http://en.wikipedia.org/wiki/Advanced_Encryption_Standard
[2] http://en.wikipedia.org/wiki/AES_implementations
[3] http://en.wikipedia.org/wiki/Advanced_Encryption_Standard_process
[4] http://en.wikipedia.org/wiki/Substitution-permutation_network

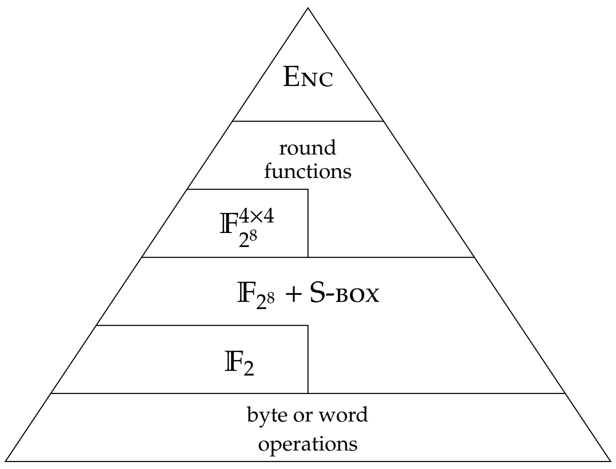


Figure 1: *The AES (encryption) arithmetic stack.*

3. AES in practice: use your understanding

The translation of theory into practice (i.e., of the AES design into a working implementation) may initially *seem* like a complex, and so daunting challenge. To address it, use of considered (versus ad hoc) a) implementation strategies and techniques plus b) development best-practices are all vitally important. The goal of this Section is to summarise these points, highlighting a variety of related decisions and implications: by doing so it supports Section 4, which presents a set of associated tasks.

3.1. Implementation strategy

At a high level, (at least) three points are useful when forming a remit and strategy for implementation; said points are somewhat generic, so it is possible they can be applied beyond the context of AES.

1. **Follow the standard.** Although abused[5] in the context of marketing and sales, the phrase "nobody ever got fired for buying IBM equipment" captures the idea that a so-called safe option (e.g., a respected brand name) may be preferable *even if* a valid, technical counterargument (e.g., the alternative is a technically superior product by an unknown vendor) exists. One could posit a paraphrased alternative "nobody ever got fired for following the standard" for cryptographic implementation tasks: using FIPS-197 [1] as a reference for AES is useful, for example, because 1) by definition it can (and so should) be treated as the definitive specification for an implementation, 2) even if that specification is flawed (e.g., AES turns out to be insecure) it offers shared risk and liability, 3) it ensures consistency and inter-operability (in terms of both functionality *and* communication, e.g., notation) with other implementations.
2. **Only implement the functionality you need.** To limit the remit of an implementation, it could be reasonable to simplify the functionality we intend it to offer. For example, we could

   (a) support only AES-128 (i.e., a 128-bit key size),
   (b) support only encryption (versus decryption) in only ECB (so not, e.g., CBC) mode, and
   (c) limit the design space, fixing a platform and so constraints for the implementation: by fixing use of a constrained platform (e.g., an ARM-based micro-controller), for example, we focus on trade-offs that favour memory footprint over efficiency.

   Of course the resulting implementation might only then be useful in specific (versus general) contexts, due to various limitations, but 1) this context may be reasonable, meaning the implementation is already useful, and, even then, 2) it affords a starting point that can later be embellished (e.g., generalised or extended) to suit.
3. **Work step-by-step rather than all-or-nothing.** By using the implementation strategy as a guide, we then decompose the implementation into simpler components. The obvious way to do so is with reference to Figure 1, which illustrates the AES arithmetic stack: the layered illustration naturally hints at a step-by-step, bottom-up route toward an implementation.

Next, using the latter points as a guide, we can start to identify both the components (e.g., functions) required, plus the options available when implementing those components. AES depends fundamentally on $\mathbb{F}_{2^8}$, a finite field of $2^8 = 256$ elements. [1, Section 3 and 4] offers an overview, divided into roughly two themes that mirror the

[5] http://en.wikipedia.org/wiki/Fear,_uncertainty_and_doubt

division between challenges of representation (i.e., data structures) and computation (i.e., algorithms): we follow a similar approach in summarising pertinent points below.

Representation. In a formal sense, $\mathbb{F}_{2^8}$ is constructed as $\mathbb{F}_2[\mathbf{x}]/p(\mathbf{x})$, i.e., as a set of binary polynomials modulo the irreducible polynomial $p(\mathbf{x}) = \mathbf{x}^8 + \mathbf{x}^4 + \mathbf{x}^3 + \mathbf{x} + 1$. In practical terms, this means we represent elements of $\mathbb{F}_{2^8}$ as sequences of bits: the i-th bit, which can be either 0 or 1, represents the i-th coefficient in an associated polynomial. Using this as a starting point, and per Figure 1, two requirements can be identified and easily resolved:

- the lower layers require representation of field elements in $\mathbb{F}_{2^8}$; this is realised using an unsigned 8-bit integer, or byte, data type (in C, e.g., `uint8_t`), and
- the upper layers require representation of $\mathbb{F}_{2^8}^{4\times4}$, i.e., (4×4)-element state and round key matrices; this is realised using a 16-element (flat, column-major) array of field elements (in C, e.g., `uint8_t s[ 16 ]` or analogous pointer `uint8_t* s`).

Computation. The requirements for computation can be divided in a similar way: per Figure 1, an implementation of AES is required to perform 1) operations on representations of $\mathbb{F}_{2^8}$, i.e., arithmetic in $\mathbb{F}_{2^8}$, including application of the S-box, plus 2) operations on representations of $\mathbb{F}_{2^8}^{4\times4}$, e.g., the AES round functions; we use these components as building blocks, for example to realise the required key expansion and encryption functionality.

- Certain (arithmetic) operations in $\mathbb{F}_{2^8}$ are trivial, due to our representation and capabilities of the underlying platform. Consider, for example, that for $x, y \in \mathbb{F}_{2^8}$ we have that

$$x + y \;\equiv\; x - y \;\equiv\; x_i + y_i \pmod 2 \;\mapsto\; x \oplus y.$$

  Put another way, addition (resp. subtraction) in the finite field is equivalent to XOR'ing the representations together (because XOR acts like a carry-less, 1-bit addition, which matches addition in $\mathbb{F}_2$).
- Likewise, the limited range and domain of (arithmetic) operations in $\mathbb{F}_{2^8}$ suggests potential for pre-computation: although careful analysis of constraints on memory footprint is important, it is plausible to (offline) pre-compute a look-up table of results and so avoid (online) computation. The S-box[6] is a good example: this is essentially a function

$$\textsc{S-box} : \mathbb{F}_{2^8} \longrightarrow \mathbb{F}_{2^8},$$

  so has 256 possible (8-bit) inputs and 256 possible (8-bit) outputs. As a result, for the overhead of 256B of memory we can simply look-up results rather than compute them; clearly this will be significantly easier, *and* imply lower latency.
- Given the ability to perform operations in $\mathbb{F}_{2^8}$, the AES round functions, namely, `AddRoundKey`, `SubBytes`, `ShiftRows`, and `MixColumns`, are reasonably simple to implement. Each such function can operate on a (partially processed) state matrix in-place[7], which aligns with the goal of minimising memory footprint.
- Alongside the round functions, computation of round keys (as used by `AddRoundKey`) from the cipher key is important. AES-128 requires a total of 11 such round keys, the 0-th instance of which is $rk^{(0)} = k$, i.e., matches the cipher key; each next, $(i+1)$-th instance $rk^{(i+1)}$ for $0 < r < 11$ is produced as a function of the previous, i-th instance $rk^{(i)}$. This means each encryption operation can take the cipher key as input, and evolve it in-place to produce successive round keys; versus the alternative of pre-computing said round keys, doing so aligns with the goal of minimising memory footprint.

3.2. Test strategy

Beyond implementation per se, the challenges of verification and testing are vitally important. This is true for (at least) two reasons. First, and more obviously, correctness is an important requirement. This universally true, but of specific relevance for security-critical kernels such as AES as used in high-assurance contexts. Second, and less obviously, however, design and use of an effective test strategy will support more effective debugging and thus more efficient development cycles. Focusing on testing, a variety of generic strategies can be considered: although only representative of the much broader discipline[8], we present some examples below. Note that each has positive and negative features, so are most sensible used and combined to suit both 1) the function under test (or being tested), plus 2) the goal and any constraints (e.g., time or space). Also note that although the descriptions focus on AES encryption as the function under test, this is not a limitation: they can be (selectively) employed in other cases, such as arithmetic in $\mathbb{F}_{2^8}$, as well.

[6] http://en.wikipedia.org/wiki/Rijndael_S-box
[7] http://en.wikipedia.org/wiki/In-place_algorithm
[8] http://en.wikipedia.org/wiki/Software_testing

1. **Use known-good test vectors.** [1, Appendix B] includes a worked example based on the test vector (k, m, c) where

$$\begin{array}{lcl} k & = & \langle\, 2B, 7E, 15, 16, 28, AE, D2, A6, AB, F7, 15, 88, 09, CF, 4F, 3C \,\rangle_{(2^8)} \\ m & = & \langle\, 32, 43, F6, A8, 88, 5A, 30, 8D, 31, 31, 98, A2, E0, 37, 07, 34 \,\rangle_{(2^8)} \\ c & = & \langle\, 39, 25, 84, 1D, 02, DC, 09, FB, DC, 11, 85, 97, 19, 6A, 0B, 32 \,\rangle_{(2^8)} \end{array}$$

   Each such test vector asserts that

$$\text{AES-128.Enc}(k, m) = c$$

   or, conversely,

$$\text{AES-128.Dec}(k, c) = m,$$

   and, given their source, we can be confident these equalities are accepted as correct (i.e., are known-good).

   Such test vectors are often provided as part of a design, so that any associated implementation can be tested. To do so, for each test vector we 1) use the implementation under test to encrypt m (resp. decrypt c) under k, computing a ciphertext (resp. plaintext) t, then 2) compare the LHS t with the known-good RHS c (resp. m), e.g.,

$$t = \text{AES-128.Enc}(k, m) \overset{?}{=} c.$$

   If the LHS and RHS match, we gain confidence that the implementation is correct; if not, we can attempt to debug it (noting the worked example includes all relevant intermediate values, which can act as further known-good references). Use of test vectors also has an advantage, in the sense they can be intelligently selected so as to target some feature; an example might be to satisfy demands on test coverage[9] that would be more difficult within a randomised approach.

2. **Use a known-good reference implementation (or oracle).** An obvious alternative is to compare the implementation with some existing alternative; the latter is often termed a reference implementation or oracle[10]. A central advantage of doing so is the lack of dependency on (and so need to supply) any particular test vector(s).

   As motivated by Section 1, this approach is made easier by the ubiquity of AES: almost any software library can act as a viable reference implementation. Denoting the reference implementation by $\mathcal{O}$, the idea is to test whether

$$\text{AES-128.Enc}(k, m) \overset{?}{=} \mathcal{O}(k, m).$$

   That is, we 1) select random k and m, 2) compute the LHS using the implementation under test, 3) compute the RHS using the reference implementation, 4) test whether the LHS equals the RHS. Put another way, the reference implementation could be viewed as generating the known-good c (resp. m) on demand; although the process differs, we basically generate then use a test vector $(k, m, c = \mathcal{O}(k, m))$. However, note that versus a fixed, finite number of provided test vectors, this approach can be used to improve our confidence by simply repeating as many (randomised) iterations as required.

3. **Use axiomatic "self-test".** Finally, *some* functionality affords a form of "self-test" in the sense that the result can be tested using an axiom (which is assumed correct by definition). Consider the axiom

$$\text{AES-128.Dec}(k, \text{AES-128.Enc}(k, m)) \overset{?}{=} m,$$

   which, intuitively, captures the fact Enc and Dec should act as inverses under k for any "correct" block cipher. The idea then, is as above: we a) select random k and m, b) compute the LHS using the implementation under test, c) test whether the LHS equals the RHS.

   Again, use of randomised inputs offers an advantage by allowing us to easily improve our confidence. In addition, this approach allows removal of the dependency on a reference implementation; doing so could, for example, be useful in supporting the use of online (versus offline) testing. However, as a counterargument, we now need some additional functionality, i.e., Dec, that we previously did not.

4. Some hands-on tasks and challenges

4.1. Build, execute, and experiment with an AES reference implementation

The material provided for this lab. worksheet includes a set of examples. Each one uses a different programming language to perform AES encryption using an associated support library (i.e., AES reference implementation); each one uses the test vector cited in Section 3.2 as a (limited) way to verify correctness.

1. Fix the working directory:

```
cd ${ARCHIVE}
```

[9] http://en.wikipedia.org/wiki/Code_coverage
[10] http://en.wikipedia.org/wiki/Oracle_(software_testing)

2. Build any material related to the example:
   ```
   make all
   ```
3. Execute *either* the Python-based[11]
   ```
   python3 encrypt.py
   ```
   or the C-based
   ```
   ./encrypt
   ```
   example, noting that each one uses [1, Appendix B] as a test of correctness.

4.2. Develop your own AES implementation

By using one of the examples from the previous Section as a starting point, this Section tasks you with development of your own AES implementation. By selecting an example, you clearly select a programming language as well: although selection of C is assumed, leading to use of C-specific syntax, *any* selection is reasonable.

1. Implement and test a function
   ```
   uint8_t xtimes( uint8_t x )
   ```
   that computes $r = \mathbf{x} \otimes_{\mathbb{F}_{2^8}} x \equiv \mathbf{x} \cdot x(\mathbf{x}) \pmod{p(\mathbf{x})}$ for $x \in \mathbb{F}_{2^8}$ (i.e., multiplies x by the indeterminate $\mathbf{x}$).
2. Implement and test a function
   ```
   uint8_t aes_sbox( uint8_t x )
   ```
   that computes $r = \text{S-BOX}(x)$ for $x \in \mathbb{F}_{2^8}$ (i.e., applies the AES S-box to x).
3. Implement and test a function
   ```
   uint8_t aes_rcon( int i )
   ```
   that computes $r = \mathbf{x}^{i-1} \pmod{p(\mathbf{x})}$ (i.e., computes the i-th round constant).
4. Using [1, Section 5.2] as a guide, implement and test a function
   ```
   void aes_enc_key_evolve( uint8_t* r, const uint8_t* rk, uint8_t rc )
   ```
   that takes a current, i-th AES-128 round key matrix `rk` and a round constant `rc` as input, and operates on it to compute a next, $(i+1)$-th AES-128 round key matrix `r` as output.
5. Using [1, Section 5.1.1-4] as a guide, implement and test
   (a) a function
   ```
   void aes_enc_rnd_key( uint8_t* s, const uint8_t* rk )
   ```
   relating to the round function `AddRoundKey`,
   (b) a function
   ```
   void aes_enc_rnd_sub( uint8_t* s )
   ```
   relating to the round function `SubBytes`,
   (c) a function
   ```
   void aes_enc_rnd_row( uint8_t* s )
   ```
   relating to the round function `ShiftRows`, and
   (d) a function
   ```
   void aes_enc_rnd_mix( uint8_t* s )
   ```
   relating to the round function `MixColumns`.

   Each function takes a current, i-th AES-128 state matrix `s` (plus, in the first case, an AES-128 round key matrix `rk`) as input, and operates on it in-place to compute a next, $(i+1)$-th AES-128 state matrix as output.
6. Using [1, Section 5.1] as a guide, implement and test a function
   ```
   void aes_enc( uint8_t* c, const uint8_t* m, const uint8_t* k )
   ```
   that takes an AES-128 plaintext `m` and AES-128 cipher key `k`, as input, and computes an AES-128 ciphertext `c` as output.

[11] Depending on the platform you have opted to use, satisfying the dependencies required *may* demand use of a Python virtual environment; Appendix A offers an overview of how to do so.

4.3. Extend your own AES implementation

1. This lab. worksheet has focused exclusively on AES *encryption*, i.e., on computation of

$$c = \text{AES-128.E}\textsc{nc}(k, m).$$

   Although this simplification might seem restrictive, when used in CTR mode [**nist:sp.800.38a**], e.g.,

$$\begin{array}{rcl} c[i] & = & m[i] \oplus \text{AES-128.E}\textsc{nc}(k, i) \\ m[i] & = & c[i] \oplus \text{AES-128.E}\textsc{nc}(k, i) \end{array}$$

   it provides the functionality required to both encrypt plaintexts *and* decrypt ciphertexts.

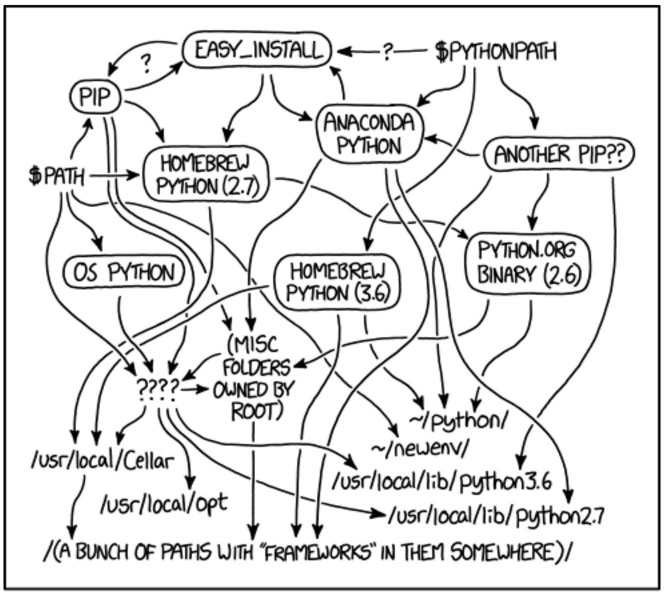
   That said, however, offering standalone decryption functionality *is* important in other use-cases. To support doing so, AES decryption is outlined by [1, Section 5.3]: it essentially describes how each encryption step is inverted. Following a similar, step-by-step approach as for AES encryption in the tasks above, develop an AES decryption implementation.
2. This lab. worksheet has focused exclusively on one platform, and one implementation strategy for AES on it. For a different platform or use-cases, however, such a focus may be inappropriate: an obvious example is whenever a trade-off that favours efficiency over memory footprint is required, e.g., for applications such as encryption of network traffic.

   With this in mind, [1, Section 6.4] offers some coverage of other implementation strategies; it includes use of T-tables, for example, as was explained in the lecture slot(s). Using your implementation in the tasks above as a starting point, try to develop an alternative based on T-tables. Compare the two strategies with respect to the trade-off(s) made: concretely, 1) how much more memory does the alternative use, and 2) how much lower is the resulting execution latency?

References

[1] *Advanced Encryption Standard (AES)*. National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 197-upd1. 2023. URL: https://doi.org/10.6028/NIST.FIPS.197-upd1 (see pp. 2, 3, 5–7).

A Using a Python virtual environment



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

As alluded to by use of "dependency hell"[12] as a colloquialism, management of software dependencies can represent a significant and, at times, frustrating challenge. Set within the context of Python, the concept of a virtual environment specifically attempts to address this challenge. Working in combination with an associated package manager[13], a virtual environment is a self-contained directory structure into which packages beyond the standard library can be installed. This, for example, avoids the need for their centralised installation (which will often require a higher, e.g., super-user[14] privilege level): the entire mechanism can be user-managed.

Use of virtual environments does not solve *every* problem, as alluded to by the cartoon above; if you opt to work using a unit-specific Vagrant box then you may not even have to use them, because relevant packages are often installed centrally as part of the provisioning step. However, particularly if you opt to work using UoB-managed[15] equipment, virtual environments can offer an important solution you will need to make use of. The Python documentation[16] itself contains an accessible overview, but the general workflow can be summarised as follows:

1. Store the virtual environment path in an environment variable, so it can be referenced easily later:

```
export VENV="${PWD}/venv"
```

2. Initialise the virtual environment:

```
python3 -m venv ${VENV}
```

3. Activate the virtual environment:

```
source ${VENV}/bin/activate
```

   While the virtual environment is active, implying use of packages installed locally within it rather than some centralised installation, the indicator `(venv)` is normally included as part of the shell prompt.

4. Install packages in the virtual environment, e.g.,

```
python3 -m pip install pycryptodomex
```

5. Use the virtual environment somehow.
6. Deactivate the virtual environment:

```
deactivate
```

Note that the virtual environment itself is persistent (in `${VENV}`), but one would need to *re*activate it when using another BASH shell (or prompt, e.g., a terminal window).

[1]http://xkcd.com/1987
[12]http://en.wikipedia.org/wiki/Dependency_hell
[13]See, e.g., https://en.wikipedia.org/wiki/Pip_(package_manager)
[14]https://en.wikipedia.org/wiki/Superuser
[15]Previously, IT Services centrally installed various packages on the workstations in, e.g., MVB Linux lab(s). (e.g., MVB-1.15 or MVB-2.11). Over time, doing so on a per-unit basis become difficult to manage; this fact led to a shift of policy, meaning use of virtual environments as the default.
[16]See, e.g., https://docs.python.org/3/tutorial/venv.html