1.
2. We will prove a more general version of this statement. Recall that a *lattice* $\Lambda \subseteq \mathbb{R}^n$ is a discrete subgroup containing a basis of $\mathbb{R}^n$; cf. §A.2.1 in the lecture notes.

   If $\Lambda \subseteq \mathbb{R}^n$ is a lattice, then $\Lambda = \mathbb{Z}^n g$ for some $g \in \mathrm{GL}_n(\mathbb{R})$; cf. Lem. 3 in §A.2.1 of the lecture notes. Throughout this exercise, $\Lambda \subseteq \mathbb{R}^n$ is a lattice and $\Lambda' \leqslant \Lambda$ is a subgroup.

   a) Let $\Lambda' < \mathbb{R}^n$ be a lattice. Let F be a fundamental domain for the translation action of Λ on $\mathbb{R}^n$. Let $S \subseteq \Lambda$ be a set of representatives in Λ for the elements of Λ/Λ'. Note that S is countable since it is a subset of Λ.
   Note that
   $$\bigsqcup_{\ell \in S} F + \ell$$
   is a fundamental domain of Λ'; we skip the verification which is purely formal. Since F is a fundamental domain of Λ, the union is indeed disjoint. So we have
   $$\mathrm{covol}(\Lambda') = \mathrm{vol}(\bigsqcup_{\ell \in S} F + \ell)$$
   This formula shows that S is finite and
   $$\begin{aligned}\mathrm{covol}(\Lambda') &= \mathrm{vol}(\bigsqcup_{\ell \in S} F + \ell)\\ &= \sum_{\ell \in S} \mathrm{vol}(F + \ell)\\ &= |S|.\mathrm{vol}(F)\\ &= [\Lambda : \Lambda'].\mathrm{covol}(\Lambda)\end{aligned}$$

   b)
3. a) As $m+i \notin (p)$, we know that $\pi \neq (p)$. In particular, the map $\mathbb{Z}[i]/(p) \to \mathbb{Z}[i]/\pi$ is not injective and therefore the latter quotient has cardinality 1 or p.
   Note that the cardinality is 1 if and only if $\pi = (1)$. However, we note that for any element $x \in \pi$ we have $p|\mathrm{Nr}(x)$. To this end let $\alpha, \beta, \gamma, \delta \in \mathbb{Z}$ and note that
   $$\mathrm{Nr}\big((\alpha+\beta i)(m+i) + (\gamma+\delta i)p\big) \equiv (\alpha m - \beta)^2 + (\alpha + \beta m)^2 \equiv 0 \bmod p.$$
   But $p \nmid \mathrm{Nr}(1)$ and therefore $\pi \neq (1)$. In particular, we have $|\mathbb{Z}[i]/\pi| = p$.
   Let r be a generator of π. Then
   $$\mathrm{Nr}(r) = p$$
   and therefore r is irreducible as any non-trivial factorization of r into irreducibles implies that $\mathrm{Nr}(r)$ is composite by multiplicativity of the norm. As the proper prime ideals in $\mathbb{Z}[i]$ are exactly the ideals generated by irreducibles, it follows that π is a prime ideal.

   b) Let $q = a + bi$. As $\mathrm{Nr}(q) = p$, it follows from the multiplicativity of the norm that q is irreducible in $\mathbb{Z}[i]$. Hence it generates a prime ideal containing $p = \mathrm{Nr}(q)$. If $p|q$, then
   $$p^2 = \mathrm{Nr}(p)|\mathrm{Nr}(q) = p$$

in $\mathbb{Z}$. This is absurd. Thus either $p \nmid a$ or $p \nmid b$. After multiplying by a unit, we can assume without loss of generality that $p \nmid b$.
Let $s, t \in \mathbb{Z}$ such that $sb + tp = 1$. Then

$$(sa)^2 + 1 = \mathrm{Nr}\big(sq + tpi\big) = s^2p + (\overline{q} - q)stpi + t^2p^2$$
$$= s^2p + 2\,\mathrm{Im}(q)stp + t^2p^2 \equiv 0 \bmod p.$$

Hence, letting $m = sa$, we have that $m^2 \equiv -1 \bmod p$. Clearly $m+i = sq+tpi \in \pi$ and thus $(m+i, p) \subset \pi$. It remains to show that q is a $\mathbb{Z}[i]$-linear combination of $m + i$ and p. Indeed,

$$b(m + i) + tap = q.$$

c) We assume without loss of generality that $m \in \mathbb{N}$ is chosen so that $1 \leqslant m < p$. In particular, we have that

$$\mathrm{Nr}(m + i) = m^2 + 1 \leqslant (p - 1)^2 + 1 = p^2 - 2(p - 1) \leqslant p^2 - 2 < \mathrm{Nr}(p).$$

We claim that for any non-zero $q \in \pi$ satisfying $\mathrm{Nr}(q) < p^2$ we have that

$$q|p \implies \pi = (q).$$

Indeed, as we have already argued, we know that $\mathrm{Nr}(q) \in p\mathbb{Z}$. On the other hand, $q|p$ implies $\mathrm{Nr}(q)|p^2$ by the multiplicativity of the norm and therefore $\mathrm{Nr}(q) = p$ as q was assumed to have norm strictly less than the norm of p.
Therefore, we obtain Algorithm 1 which determines a generator of π. Indeed, by the above reasoning, at each step the norm of the remainder is a (strictly decreasing) non-zero multiple of p unless q has norm p.

d) By the preceding discussion, we know that any representative $p = \mathrm{Nr}(q)$ is associated to either r or $\overline{r}$, where $r \in \mathbb{Z}[i]$ is a generator of π.
The pseudocode of a solution can be found in Algorithm . We have implemented two versions of the code. The first (`FindReps`) solves the polynomial equation $X^2+1 = 0$ over the field $\mathbb{F}_p$ via the general implemented root finding algorithms. The second algorithm (`FindRepsRoot`) employs a square root implemented for finite fields. Finally, we have implemented a brute force algorithm which for $1 \leqslant a < p$ calculates $B = p - a^2$ and then checks whether B is a square. All the three codes can be found in Listing 1.

Algorithm 1 Compute a generator of π.

function GENERATOR(p, m)
  $q \leftarrow m + i$
  $z \leftarrow p$
  while $\mathrm{Nr}(q) \neq p$ **do**
    $z = kq + r$ with $\mathrm{Nr}(r) < \mathrm{Nr}(q)$
    $z \leftarrow q$
    $q \leftarrow r$
  end while
  return q
end function

```
# import the time module to compare running times for the
# different algorithms
import time

# PRE:  Tuples z = [a,b] and q = [c,d] of integers
# POST: Return value is a remainder r = [x,y] such that z = kq + r
```

Algorithm 2 Write p as a sum of two squares.

function FINDREPS(p)
  if $p \not\equiv 1 \bmod 4$ **then**
    $m \leftarrow$ any root of $X^2 + 1$ over $\mathbb{F}_p$
    $q \leftarrow$ GENERATOR(p,m)
    return $a = \mathrm{Re}(q), b = \mathrm{Im}(q)$
  end if
end function

```
#       for some Gaussian integer k and such that Nr(r) < Nr(q).
def Remainder(z,q):
    if (q[0] != 0 or q[1] != 0):
        n = q[0]^2 + q[1]^2
        # We define s = z/q = [u, v]
        u = (z[0]*q[0] + z[1]*q[1])/n
        v = (z[1]*q[0] - z[0]*q[1])/n
        # We find the Gaussian integer closest to s
        # This is one of the corners of the square containing s
        reals = [floor(u),ceil(u)]
        ims = [floor(v),ceil(v)]
        # We use a loop to find out which corner is closest to s
        # We save the candidate for the remainder
        remainder = [0,0]
        # We initialize the (squared) distance to the corner to
        # be 1; note that the closest corner is at (squared)
        # distance at most 1/2
        distance = 1
        for i in range(2):
            for j in range(2):
                corner = [reals[i],ims[j]]
                a = u - corner[0]
                b = v - corner[1]
                d = a^2 + b^2
                if d < distance:
                    distance = d
                    # If k is the closest corner, then r = z - k*q
                    # is a remainder for division of z with respect
                    # to q satisfying Nr(r) < Nr(q).
                    remainder = [a*q[0] - b*q[1] , a*q[1] + b*q[0]]
        return remainder
    else:
        print('Division by 0')

# PRE:  Integers m, p
# POST: Return value is a generator of the principal ideal
#       generated by m+i and p inside the ring of Gaussian
#       integers. The generator is determined using part
#       (c) of the exercise
def Generator(m,p):
    q = [m,1]
    z = [p,0]
    while q[0]^2 + q[1]^2 != p:
        r = Remainder(z,q)
        z = q
        q = r
    return q

# PRE:      Odd prime p equivalent to 1 mod 4
```

```
# POST:      Return value is the, up to permutation, unique
#            pair (a,b) of natural numbers a and b such that
#            p = a^2 + b^2
# REMARK:    This version uses the polynomial ring over a
#            finite field and the implemented root finding
#            algorithms to find a solution to X^2+1=0
def FindReps(p):
    if (p in Primes()) and (p == mod(1,4)):
        # Used for timing the algorithm.
        # Set starting time here
        tic = time.perf_counter()
        # Define the field with p elements
        k = GF(p)
        # Define the polynomial ring over k in one variable t
        R.<t> = PolynomialRing(k, 't')
        P = t^2+1
        # If x is an element in k, then x.lift() is a
        # representative of x in ZZ
        m = P.roots()[0][0].lift()
        # Find z = a+bi in the Gaussian integers satisfying N(z)=p
        # We use parts (a) and (b) of the exercise as well as
        # exercise 2 to note that such z, up to associatedness
        # and complex conjugation is given by a representative of
        # a prime ideal dividing (p)
        reps = Generator(m,p)
        s = ''.join(('Up to sign and permutation, ',
                     'every representation p=a^2+b^2 ',
                     'is of the form a=',
                     str(abs(reps[0])),
                     ' and b=',
                     str(abs(reps[1]))))
        print(s)
        # Set end time here
        toc = time.perf_counter()
        print(f"code took {toc - tic:0.4f} seconds")
        return reps
    else:
        print('Invalid input. ' +
              'Input must be a prime of residue 1 mod 4.')

# PRE:       odd prime p equivalent to 1 mod 4
# POST:      return value is the, up to permutation, unique
#            pair (a,b) of natural numbers a and b such that
#            p = a^2 + b^2
# REMARK:    This version uses the square_root function to
#            find a root of p-1 in the finite field with p
#            elements.
def FindRepsRoot(p):
    if (p in Primes()) and (p == mod(1,4)):
        # Used for timing the algorithm.
        # Set starting time here
        tic = time.perf_counter()
        # Define the field with p elements
        k = GF(p)
        # Find a square root of p-1 in k
        root = k(p-1).square_root()
        # if x is an element in k, then x.lift() is a
    representative
        # of x in ZZ
        m = root.lift()
```

```
        # Find z = a+bi in the Gaussian integers satisfying N(z)=p
        # We use parts (a) and (b) of the exercise as well as
        # exercise 2 to note that such z, up to associatedness and
        # complex conjugation is given by a representative of a
        # prime ideal dividing (p)
        reps = Generator(m,p)
        s = ''.join(('Up to sign and permutation, ',
                     'every representation p=a^2+b^2 ',
                     'is of the form a=',
                     str(abs(reps[0])),
                     ' and b=',str(abs(reps[1]))))
        print(s)
        # Set end time here
        toc = time.perf_counter()
        print(f"code took {toc - tic:0.4f} seconds")
        return reps
    else:
        print('Invalid input. ' +
              'Input must be a prime of residue 1 mod 4.')

# PRE:      odd prime p equivalen to 1 mod 4
# POST:     return value is the, up to permutation, unique
#           pair (a,b) of natural numbers a and b such that
#           p = a^2 + b^2
# REMARK:   This is a brute force algorithm which subtracts
#           a square from p and then simply checks whether
#           the difference is a square.
def FindRepsBruteForce(p):
    if (p in Primes()) and (p == mod(1,4)):
        # Used for timing the algorithm.
        # Set starting time here
        tic = time.perf_counter()
        a = 0
        found = False
        while not found:
            a += 1
            if (p-a^2).is_square():
                found = True
        b = isqrt(p-a^2)
        s = ''.join(('Up to sign and permutation, '
                     'every representation p=a^2+b^2 ',
                     'is of the form a=',str(a),
                     ' and b=',str(b)))
        print(s)
        reps = [a,b]
        # Set end time here
        toc = time.perf_counter()
        print(f"code took {toc - tic:0.4f} seconds")
        return reps
    else:
        print('Invalid input. ' +
              'Input must be a prime of residue 1 mod 4.')
```

LISTING 1. Example SageMath code for Ex. 3